Programming Language Semantics

A Rewriting Approach

Grigore Roșu

University of Illinois at Urbana-Champaign

3.6 Modular Structural Operational Semantics (MSOS)

Modular structural operational semantics (MSOS) was introduced, as its name implies, to address the non-modularity aspects of (big-step and/or small-step) SOS. As already seen in Section 3.5, there are several reasons why big-step and small-step SOS are non-modular, as well as several facets of non-modularity in general. In short, a definitional framework is *non-modular* when, in order to add a new feature to an existing language or calculus, one needs to revisit and change some or all of the already defined unrelated features. For example, recall the IMP extension with input/output in Section 3.5.2. We had to add new semantic components in the IMP configurations, both in the big-step and in the small-step SOS definitions, to hold the input/output buffers. That meant, in particular, that *all* the existing big-step and/or small-step SOS rules of IMP had to change. That was, at best, very inconvenient.

Before we get into the technicalities of MSOS, one natural question to address is why we need modularity of language definitions. One may argue that defining a programming language is a major endeavor, done once and for all, so having to go through the semantic rules many times is, after all, not such a bad idea, because it gives one the chance to find and fix potential errors in them. Here are several reasons why modularity is desirable in language definitions, in no particular order:

- Having to modify many or all rules whenever a new rule is added that modifies the structure of the configuration is actually more error prone than it may seem, because rules become heavier to read and debug; for example, one can write σ instead of σ' in a right-hand-side of a rule and a different or wrong language is defined.

- A modular semantic framework allows us to more easily reuse semantics of existing and probably already well-tested features in other languages or language extensions, thus increasing our productivity as language designers and our confidence in the correctness of the resulting language definition.

- When designing a new language, as opposed to an existing language, one needs to experiment with features and combinations of features; having to do unrelated changes whenever a new feature is added to or removed from the language burdens the language designer with boring tasks taking considerable time that could have been otherwise spent on actual interesting language design issues.

- There is an increasing number of domain-specific languages, resulting from the need to abstract away from low-level programming language details to important, domain-specific application aspects. Hence, there is a need for language design and experimentation for various domains. Moreover, domain-specific languages tend to be dynamic, being added or removed features frequently as the domain knowledge evolves. It would be nice to have the possibility to "drag-and-drop" features in one's language, such as functions, exceptions, etc.; however, modularity is crucial for that.

To our knowledge, MSOS was the first framework that explicitly recognizes the importance of modular language design and provides explicit support to achieve it in the context of SOS. Reduction semantics with evaluation contexts (see Section 3.7) was actually proposed before MSOS and also offers modularity in language semantic definitions, but its modularity comes as a consequence of a different way to propagate reductions through language constructs and not as an explicit goal that it strives to achieve.

There are both big-step and small-step variants of MSOS, but we discuss only small-step MSOS here. We actually generically call MSOS the small-step, implicitly-modular variant of MSOS (see Section 3.6.4). To bring modularity to SOS, MSOS proposes the following:

- Separate the syntax (i.e., the fragment of program under consideration) from the non-syntactic components in configurations, and treat them differently, as explained below;

- Make the transitions only relate syntax to syntax (as opposed to configurations), and hide the non-syntactic components in a transition label, as explained below;
- Encode in the transition label all the changes in the non-syntactic components of the configuration that need to be applied together with the syntactic reduction given by the transition;
- Use specialized notation in transition labels together with a discipline to refer to the various semantic components and to say that some of them stay unchanged; also, labels can be explicitly or implicitly shared by the conditions and the conclusion of a rule, elegantly capturing the idea that "changes are propagated" through desired language constructs.

A transition in MSOS is of the form

$$P \xrightarrow{\Delta} P'$$

where P and P' are programs or fragments of programs and Δ is a *label describing the semantic configuration components both before and after the transition*. Specifically, Δ is a record containing fields denoting the semantic components of the configuration. The preferred notation in MSOS for stating that in label Δ the semantic component associated to the field name field *before* the transition takes place is α is $\Delta = \{\mathsf{field} = \alpha, \ldots\}$. Similarly, the preferred notation for stating that the semantic component associated to field field *after* the transition takes place is β is $\Delta = \{\mathsf{field'} = \beta, \ldots\}$ (the field name is primed). For example, the second MSOS rule for variable assignment (when the assigned arithmetic expression is already evaluated) is (this is rule (MSOS-Asgn) in Figure 3.24):

$$x = i; \xrightarrow{\{\mathsf{state}=\sigma,\ \mathsf{state'}=\sigma[i/x],\ \ldots\}} \{\} \quad \text{if } \sigma(x) \neq \bot$$

It is easy to desugar the rule above into a more familiar SOS rule of the form:

$$\langle x = i;, \sigma \rangle \rightarrow \langle \{\}, \sigma[i/x] \rangle \quad \text{if } \sigma(x) \neq \bot$$

The above is precisely the rule (SmallStep-Asgn) in the small-step SOS of IMP (see Figure 3.15). The MSOS rule is actually more modular than the SOS one, because of the "...", which says that everything else in the configuration stays unchanged. For example, if we want to extend the language with input/output language constructs as we did in Section 3.5.2, then new semantic components, namely the input and output buffers, need to be added to the configuration. Moreover, as seen in Section 3.5.2, the SOS rule above needs to be changed into a rule of the form

$$\langle x = i;, \sigma, \omega_{in}, \omega_{out} \rangle \rightarrow \langle \{\}, \sigma[i/x], \omega_{in}, \omega_{out} \rangle \quad \text{if } \sigma(x) \neq \bot$$

where ω_{in} and ω_{out} are the input and output buffers, respectively, which stay unchanged during the variable assignment operation, while the MSOS rule does not need to be touched.

To impose a better discipline on the use of labels, at the same time making the notation even more compact, MSOS splits the fields into three categories: *read-only*, *read-write*, and *write-only*. The field state above was read-write, meaning that the transition label can both read its value before the transition takes place and write its value after the transition takes place. Unlike the state, which needs to be both read and written, there are semantic configuration components that only need to be read, as well as ones that only need to be written. In these cases, it is recommended to use read-only or write-only fields.

Read-only fields are only inspected by the rule, but not modified, so they only appear unprimed in labels. For example, the following can be one of the MSOS rules for the `let` binding language construct in a pure

functional language where expressions yield no side-effects:

$$\frac{e_2 \xrightarrow{\{\text{env}=\rho[v_1/x],\ldots\}} e_2'}{\texttt{let } x = v_1 \texttt{ in } e_2 \xrightarrow{\{\text{env}=\rho,\ldots\}} \texttt{let } x = v_1 \texttt{ in } e_2'}$$

Indeed, transitions do not modify the environment in a pure functional language. They only use it in a read-only fashion to lookup the variable values. A new environment is created in the premise of the rule above to reduce the body of the `let`, but neither of the transitions in the premise nor in the conclusion of the rule change their environment. Note that this was not the case for IMP extended with `let` in Section 3.5.5, because there we wanted blocks and local variable declarations to desugar to `let` statements. Since we allowed variables to be assigned new values in blocks, like in conventional imperative and object-oriented languages, we needed an impure variant of `let`. As seen in Section 3.6.2, our MSOS definition of IMP++'s `let` uses a read-write attribute (the state attribute). We do not discuss read-only fields any further here.

Write-only fields are used to record data that is not analyzable during program execution, such as the output or the trace. Their names are always primed and they have a free monoid semantics—everything written on them is actually added to the end (see [53] for technical details). Consider, for example, the extension of IMP with an output (or print) statement in Section 3.5.2, whose MSOS second rule (after the argument is evaluated, namely rule (MSOS-PRINT) in Section 3.6.2) is:

$$\texttt{print(}i\texttt{);} \xrightarrow{\{\text{output'}=i,\ \ldots\}} \{\}$$

Compare the rule above with the one below, which uses a read-write attribute instead:

$$\texttt{print(}i\texttt{);} \xrightarrow{\{\text{output}=\omega_{out},\ \text{output'}=\omega_{out}:i,\ \ldots\}} \{\}$$

Indeed, mentioning the ω_{out} like in the second rule above is unnecessary, error-prone (e.g., one may forget to add it to the primed field or may write $i : \omega_{out}$ instead of $\omega_{out} : i$), and non-modular (e.g., one may want to change the monoid construct, say to write $\omega_{out} \cdot i$ instead of $\omega_{out} : i$, etc.).

MSOS achieves modularity in two ways:

1. By making intensive use of the record comprehension notation "...", which, as discussed, indicates that more fields could follow but that they are not of interest. In particular, if the MSOS rule has no premises, like in the rules for the assignment and print statements discussed above, than the "..." says that the remaining contents of the label stays unchanged after the application of the transition; and

2. By reusing the same label or portion of label both in the premise and in the conclusion of an MSOS proof system rule. In particular, if "..." is used in the labels of both the premise and the conclusion of an MSOS rule, then all the occurrences of "..." stand for the same portion of label, that is, the same fields bound to the same semantic components.

For example, the following MSOS rules for first-statement reduction in sequential composition are equivalent

and say that all the changes generated by reducing s_1 to s_1' are propagated when reducing $s_1\ s_2$ to $s_1'\ s_2$:

$$\frac{s_1 \xrightarrow{\Delta} s_1'}{s_1\ s_2 \xrightarrow{\Delta} s_1'\ s_2}$$

$$\frac{s_1 \xrightarrow{\{...\}} s_1'}{s_1\ s_2 \xrightarrow{\{...\}} s_1'\ s_2}$$

$$\frac{s_1 \xrightarrow{\{\mathsf{state}=\sigma,\ ...\}} s_1'}{s_1\ s_2 \xrightarrow{\{\mathsf{state}=\sigma,\ ...\}} s_1'\ s_2}$$

Indeed, advancing the first statement in a sequential composition of statements one step has the same effect on the configuration as if the statement was advanced the same one step in isolation, without the other statement involved; said differently, the side effects are all properly propagated.

MSOS (the implicitly-modular variant of it, see Section 3.6.4) has been refined to actually allow for dropping such redundant labels like above from rules. In other words, if a label is missing from a transition then the *implicit label* is assumed: if the rule is unconditional then the implicit label is the identity label (in which the primed fields have the same values as the corresponding unprimed ones, etc.), but if the rule is conditional then the premise and the conclusion transitions share the same label, that is, they perform the same changes on the semantic components of the configuration. With this new notational convention, the most elegant and compact way to write the rule above in MSOS is:

$$\frac{s_1 \rightarrow s_1'}{s_1\ s_2 \rightarrow s_1'\ s_2}$$

This is precisely the rule (MSOS-Seq-Arg1) in Figure 3.24, part of the MSOS semantics of IMP.

One of the important merits of MSOS is that it captures formally many of the tricks that language designers informally use to avoid writing awkward and heavy SOS definitions.

Additional notational shortcuts are welcome in MSOS if properly explained and made locally rigorous, without having to rely on other rules. For example, the author of MSOS finds the rule

$$x \xrightarrow{\{\mathsf{state},\ ...\}} \mathsf{state}(x) \quad \text{if } \mathsf{state}(x) \neq \bot$$

to be an acceptable variant of the lookup rule:

$$x \xrightarrow{\{\mathsf{state}=\sigma,\ ...\}} \sigma(x) \quad \text{if } \sigma(x) \neq \bot$$

despite the fact that, strictly speaking, $\mathsf{state}(x)$ does not make sense by itself (recall that state is a field name, not the state) and that field names are expected to be paired with their semantic components in labels. Nevertheless, there is only one way to make sense of this rule, namely to replace any use of state by its semantic contents, which therefore does not need to be mentioned.

A major goal when using MSOS to define languages or calculi is to write on the labels as little information as possible and to use the implicit conventions for the missing information. That is because everything written on labels may work against modularity if the language is later on extended or simplified. As an extreme case,

if one uses only read/write fields in labels and mentions all the fields together with all their semantic contents on every single label, then MSOS becomes conventional SOS and therefore suffers from the same limitations as SOS with regards to modularity.

Recall the rules in Figure 3.16 for deriving the transitive closure $\rightarrow^\star$ of the small-step SOS relation $\rightarrow$. In order for two consecutive transitions to compose, the source configuration of the second had to be identical to the target configuration of the first. A similar property must also hold in MSOS, otherwise one may derive inconsistent computations. This process is explained in MSOS by making use of category theory (see [53] for technical details on MSOS; see Section 2.7 for details on category theory), associating MSOS labels with morphisms in a special category and then using the morphism composition mechanism of category theory.

However, category theory is not needed in order to understand how MSOS works in practice. A simple way to explain its label composition is by translating, or desugaring MSOS definitions into SOS, as we implicitly suggested when we discussed the MSOS rule for variable assignment above. Indeed, once one knows all the fields in the labels, which happens once a language definition is complete, one can automatically associate a standard small-step SOS definition to the MSOS one by replacing each MSOS rule with an SOS rule over configurations including, besides the syntactic contents, the complete semantic contents extracted from the notational conventions in the label. The resulting SOS configurations will not have fields anymore, but will nevertheless contain all the semantic information encoded by them. For example, in the context of a language containing only a state and an output buffer as semantic components in its configuration (note that IMP++ contained an input buffer as well), the four rules discussed above for variable assignment, output, sequential composition, and lookup desugar, respectively, into the following conventional SOS rules:

$$\langle x = i\,;, \sigma, \omega\rangle \rightarrow \langle \{\}, \sigma[i/x], \omega\rangle \quad \text{if } \sigma(x) \neq \bot$$

$$\langle \texttt{print(}i\texttt{)}\,;, \sigma, \omega\rangle \rightarrow \langle \{\}, \sigma, \omega : i\rangle$$

$$\frac{\langle s_1, \sigma, \omega\rangle \rightarrow \langle s_1', \sigma', \omega'\rangle}{\langle s_1\ s_2, \sigma, \omega\rangle \rightarrow \langle s_1'\ s_2, \sigma', \omega'\rangle}$$

$$\langle x, \sigma, \omega\rangle \rightarrow \langle \sigma(x), \sigma, \omega\rangle \quad \text{if } \sigma(x) \neq \bot$$

Recall that for unconditional MSOS rules the meaning of the missing label fields is "stay unchanged", while in the case of conditional rules the meaning of the missing fields is "same changes in conclusion as in the premise". In order for all the changes explicitly or implicitly specified by MSOS rules to apply, one also needs to provide an initial state for all the attributes, or in terms of SOS, an initial configuration. The initial configuration is often left unspecified in MSOS or SOS paper language definitions, but it needs to be explicitly given when one is concerned with executing the semantics. In our SOS definition of IMP in Section 3.3.2 (see Figure 3.15), we created the appropriate initial configuration in which the top-level statement was executed using the proof system itself, more precisely the rule (SmallStep-Pgm) created a configuration holding a statement and a state from a configuration holding only the program. That is not possible in MSOS, because MSOS assumes that the structure of the label record does not change dynamically as the rules are applied. Instead, it assumes all the attributes given and fixed. Therefore, one has to explicitly state the initial values corresponding to each attribute in the initial state. However, in practice those initial values are understood and, consequently, we do not bother defining them. For example, if an attribute holds a list or a set, then its initial value is the empty list or set; if it holds a partial function, then its initial value is the partial function undefined everywhere; etc.

This way of regarding MSOS as a convenient front-end to SOS also supports the introduction of further

notational conventions in MSOS if desired, like the one discussed above using $\mathsf{state}(x)$ instead of $\sigma(x)$ in the right-hand-side of the transition, provided that one explains how such conventions are desugared when going from MSOS to SOS. Finally, the translation of MSOS into SOS also allows MSOS to borrow from SOS the reflexive/transitive closure $\rightarrow^\star$ of the one-step relation.

3.6.1 The MSOS of IMP

Figures 3.23 and 3.24 show the MSOS definition of IMP. There is not much to comment on the MSOS rules in these figures, except, perhaps, to note how compact and elegant they are compared to the corresponding SOS definition in Figures 3.14 and 3.15. Except for the three rules (MSOS-Lookup), (MSOS-Asgn), and (MSOS-Var), which make use of labels, they are as compact as they can be in any SOS-like setting for any language including the defined constructs. Also, the above-mentioned three rules only mention those components from the labels that they really need, so they allow for possible extensions of the language, like the IMP++ extension in Section 3.5.

The rule (MSOS-Var) is somehow different from the other rules that need the information in the label, in that it uses an attribute which has the type read-write but it only writes it without reading it. This is indeed possible in MSOS. The type of an attribute cannot be necessarily inferred from the way it is used in some of the rules, and not all rules must use the same attribute in the same way. One should explicitly clarify the type of each attribute before one gives the actual MSOS rules, and one is not allowed to change the attribute types dynamically, during derivations. Indeed, if the type of the output attribute in the MSOS rules for output above (and also in Section 3.6.2) were read-write, then the rules would wrongly imply that the output buffer will only store the last value, the previous ones being lost (this could be a desirable semantics in some cases).

Since the MSOS proof system in Figures 3.23 and 3.24 translates, following the informal procedure described above, in the SOS proof system in Figures 3.14 and 3.15, basically all the small-step SOS intuitions and discussions for IMP in Section 3.3.2 carry over here almost unchanged. In particular:

Definition 22. *We say that $C \rightarrow C'$ is derivable with the MSOS proof system in Figures 3.23 and 3.24, written* $\text{MSOS(IMP)} \vdash C \rightarrow C'$*, iff* $\text{SmallStep(IMP)} \vdash C \rightarrow C'$ *(using the proof system in Figures 3.14 and 3.15). Similarly,* $\text{MSOS(IMP)} \vdash C \rightarrow^\star C'$ *iff* $\text{SmallStep(IMP)} \vdash C \rightarrow^\star C'$.

Note, however, that MSOS is more syntactic in nature than SOS, in that each of its reduction rules requires syntactic terms in both sides of the transition relation. In particular, that means that, unlike in SOS (see Exercise 68), in MSOS one does not have the option to dissolve statements from configurations anymore. Instead, one needs to reduce them to {} or some similar syntactic constant; if the original language did not have such a constant then one needs to invent one and add it to the original language or calculus syntax.

3.6.2 The MSOS of IMP++

We next discuss the MSOS of IMP++, playing the same language design scenario as in Section 3.5: we first add each feature separately to IMP, as if that feature was the final extension of the language, and then we add all the features together and investigate the modularity of the resulting definition as well as possibly unexpected feature interactions.

Variable Increment

In MSOS, one can define the increment modularly:

$$++\,x \xrightarrow{\{\mathsf{state}=\sigma,\ \mathsf{state'}=\sigma[(\sigma(x)+_{Int}1)/x],\ ...\}} \sigma(x) +_{Int} 1 \qquad \text{(MSOS-Inc)}$$

$$x \xrightarrow{\{\text{state}=\sigma,\ ...\}} \sigma(x) \quad \text{if } \sigma(x) \neq \bot \qquad \text{(MSOS-Lookup)}$$

$$\frac{a_1 \to a_1'}{a_1 + a_2 \to a_1' + a_2} \qquad \text{(MSOS-Add-Arg1)}$$

$$\frac{a_2 \to a_2'}{a_1 + a_2 \to a_1 + a_2'} \qquad \text{(MSOS-Add-Arg2)}$$

$$i_1 + i_2 \to i_1 +_{Int} i_2 \qquad \text{(MSOS-Add)}$$

$$\frac{a_1 \to a_1'}{a_1 \ / \ a_2 \to a_1' \ / \ a_2} \qquad \text{(MSOS-Div-Arg1)}$$

$$\frac{a_2 \to a_2'}{a_1 \ / \ a_2 \to a_1 \ / \ a_2'} \qquad \text{(MSOS-Div-Arg2)}$$

$$i_1 \ / \ i_2 \to i_1 \ /_{Int} \ i_2 \quad \text{if } i_2 \neq 0 \qquad \text{(MSOS-Div)}$$

$$\frac{a_1 \to a_1'}{a_1 \ \texttt{<=}\ a_2 \to a_1' \ \texttt{<=}\ a_2} \qquad \text{(MSOS-Leq-Arg1)}$$

$$\frac{a_2 \to a_2'}{i_1 \ \texttt{<=}\ a_2 \to i_1 \ \texttt{<=}\ a_2'} \qquad \text{(MSOS-Leq-Arg2)}$$

$$i_1 \ \texttt{<=}\ i_2 \to i_1 \leq_{Int} i_2 \qquad \text{(MSOS-Leq)}$$

$$\frac{b \to b'}{\texttt{!}\ b \to \texttt{!}\ b'} \qquad \text{(MSOS-Not-Arg)}$$

$$\texttt{!}\ \texttt{true} \to \texttt{false} \qquad \text{(MSOS-Not-True)}$$

$$\texttt{!}\ \texttt{false} \to \texttt{true} \qquad \text{(MSOS-Not-False)}$$

$$\frac{b_1 \to b_1'}{b_1 \ \texttt{\&\&}\ b_2 \to b_1' \ \texttt{\&\&}\ b_2} \qquad \text{(MSOS-And-Arg1)}$$

$$\texttt{false}\ \texttt{\&\&}\ b_2 \to \texttt{false} \qquad \text{(MSOS-And-False)}$$

$$\texttt{true}\ \texttt{\&\&}\ b_2 \to b_2 \qquad \text{(MSOS-And-True)}$$

Figure 3.23: MSOS(IMP) — MSOS of IMP Expressions ($i_1, i_2 \in \mathit{Int}$; $x \in \mathit{Id}$; $a_1, a_1', a_2, a_2' \in \mathit{AExp}$; $b, b', b_1, b_1', b_2 \in \mathit{BExp}$; $\sigma \in \mathit{State}$).

$$\frac{a \to a'}{x = a; \to x = a';} \qquad \text{(MSOS-Asgn-Arg2)}$$

$$x = i; \xrightarrow{\{\text{state}=\sigma,\ \text{state'}=\sigma[i/x],\ ...\}} \{\} \quad \text{if } \sigma(x) \neq \bot \qquad \text{(MSOS-Asgn)}$$

$$\frac{s_1 \to s_1'}{s_1\ s_2 \to s_1'\ s_2} \qquad \text{(MSOS-Seq-Arg1)}$$

$$\{\}\ s_2 \to s_2 \qquad \text{(MSOS-Seq-Skip)}$$

$$\frac{b \to b'}{\texttt{if}\,(b)\ s_1\ \texttt{else}\ s_2 \to \texttt{if}\,(b')\ s_1\ \texttt{else}\ s_2} \qquad \text{(MSOS-If-Arg1)}$$

$$\texttt{if}\,(\texttt{true})\ s_1\ \texttt{else}\ s_2 \to s_1 \qquad \text{(MSOS-If-True)}$$

$$\texttt{if}\,(\texttt{false})\ s_1\ \texttt{else}\ s_2 \to s_2 \qquad \text{(MSOS-If-False)}$$

$$\texttt{while}\,(b)\ s \to \texttt{if}\,(b)\,\{\,s\ \texttt{while}\,(b)\ s\,\}\,\texttt{else}\,\{\} \qquad \text{(MSOS-While)}$$

$$\texttt{int}\ xl;\ s \xrightarrow{\{\text{state'}=\ xl\mapsto 0,\ ...\}} s \qquad \text{(MSOS-Var)}$$

Figure 3.24: MSOS(IMP) — MSOS of IMP Statements ($i \in \mathit{Int}$; $x \in \mathit{Id}$; $xl \in \mathbf{List\{}\mathit{Id}\mathbf{\}}$; $a, a' \in \mathit{AExp}$; $b, b' \in \mathit{BExp}$; $s, s_1, s_1', s_2 \in \mathit{Stmt}$; $\sigma \in \mathit{State}$).

No other rule needs to be changed, because MSOS already assumes that, unless otherwise specified, each rule propagates all the configuration changes in its premise(s).

Input/Output

MSOS can modularly support the input/output extension of IMP. We need to add new label attributes holding the input and the output buffers, say input and output, respectively, and then to add the corresponding rules for the input/output constructs. Note that the input attribute is read-write, while the output attribute is write-only. Here are the MSOS rules for input/output:

$$\texttt{read()} \xrightarrow{\{\text{input}=i:\omega,\ \text{input'}=\omega,\ ...\}} i \qquad \text{(MSOS-Read)}$$

$$\frac{a \to a'}{\texttt{print}(a);\ \to \texttt{print}(a');} \qquad \text{(MSOS-Print-Arg)}$$

$$\texttt{print}(i);\ \xrightarrow{\{\text{output'}=i,\ ...\}} \{\} \qquad \text{(MSOS-Print)}$$

Note that, since output is a write-only attribute, we only need to mention the new value that is added to the output in the label of the second rule above. If output was declared as a read-write attribute, then the label of the second rule above would have been $\{\text{output} = \omega, \text{output'} = \omega : i, \ldots\}$. A major implicit objective of MSOS is to minimize the amount of information that the user needs to write in each rule. Indeed, anything written by a user can lead to non-modularity and thus work against the user when changes are performed to

the language. For example, if for some reason one declared output as a read-write attribute and then later on one decided to change the list construct for the output integer list from colon "_ : _" to something else, say "_ · _", then one would need to change the label in the second rule above from {output = ω, output' = $\omega : i, \ldots$} to {output = ω, output' = $(\omega \cdot i), \ldots$}. Therefore, in the spirit of enhanced modularity and clarity, the language designer using MSOS is strongly encouraged to use write-only (or read-only) attributes instead of read-write attributes whenever possible.

Notice the lack of expected duality between the rules (MSOS-Read) and (MSOS-Print) for input and for output above. Indeed, for all the reasons mentioned above, one would like to write the rule (MSOS-Read) more compactly and modularly as follows:

$$\texttt{read()} \xrightarrow{\{\text{input}=i,\ \ldots\}} i$$

Unfortunately, this is not possible with the current set of label attributes provided by MSOS. However, there is no reason why MSOS could not be extended to include more attributes. For example, an attribute called "consumable" which would behave as the dual of write-only, i.e., it would only have an unprimed variant in the label holding a monoid (or maybe a group?) structure like the read-only attributes but it would consume from it whatever is matched by the rule label, would certainly be very useful in our case here. If such an attribute type were available, then our input attribute would be of that type and our MSOS rule for `read()` would be like the one above.

A technical question regarding the execution of the resulting MSOS definition is how to provide input to programs. Or, put differently, how to initialize configurations. One possibility is to assume that the user is fully responsible for providing the initial attribute values. This is, however, rather inconvenient, because the user would then always have to provide an empty state and an empty output buffer in addition to the desired input buffer in each configuration. A more convenient approach is to invent a special syntax allowing the user to provide precisely a program and an input to it, and then to automatically initialize all the attributes with their expected values. Let us pair a program p and an input ω for it using a configuration-like notation of the form $\langle p, \omega \rangle$. Then we can replace the rule (MSOS-Var) in Figure 3.24 with the following rule:

$$\langle \texttt{int}\ xl\texttt{;}\ s,\ \omega \rangle \xrightarrow{\{\text{state'}=xl\mapsto 0,\ \text{input'}=\omega,\ \ldots\}} s$$

Abrupt Termination

MSOS allows for a more modular semantics of abrupt termination than the more conventional semantic approaches discussed in Section 3.5.3. However, in order to achieve modularity, we need to extend the syntax of IMP with a `top` construct, similarly to the small-step SOS variant discussed in Section 3.5.3. The key to modularity here is to use the labeling mechanism of MSOS to carry the information that a configuration is in a halting status. Let us assume an additional write-only field in the MSOS labels, called halting, which is *true* whenever the program needs to halt, otherwise it is *false*[7]. Then we can add the following two MSOS rules that set the halting field to *true*:

$$i_1 \mathbin{/} 0 \xrightarrow{\{\text{halting}'=true,\ \ldots\}} i_1 \mathbin{/} 0 \qquad \text{(MSOS-Div-By-Zero)}$$

$$\texttt{halt;} \xrightarrow{\{\text{halting}'=true,\ \ldots\}} \texttt{halt;} \qquad \text{(MSOS-Halt)}$$

[7] Strictly speaking, MSOS requires that the write-only attributes take values from a free monoid; if one wants to be faithful to that MSOS requirement, then one can replace *true* with some letter word and *false* with the empty word.

As desired, it is indeed the case now that a sequent of the form $s \xrightarrow{\{\text{halting}'=\mathit{true},\ ...\}} s'$ is derivable if and only if $s = s'$ and the next executable step in s is either a "`halt;`" statement or a division-by-zero expression. If one does not like keeping the syntax unchanged when an abrupt termination takes place, then one can add a new syntactic construct, say `stuck` like in [53], and replace the right-hand-side configurations above with `stuck`; that does not conceptually change anything in what follows. The setting seems therefore perfect for adding a rule of the form

$$\frac{s \xrightarrow{\{\text{halting}'=\mathit{true},\ ...\}} s}{s \xrightarrow{\{\text{halting}'=\mathit{false},\ ...\}} \{\}}$$

and declare ourselves done, because now an abruptly terminated statement terminates just like any other statement, with a {} statement as result and with a label containing a non-halting status. Unfortunately, that does not work, because such a rule would interfere with other rules taking statement reductions as preconditions, for example with the first precondition of the (MSOS-SEQ) rule, and thus hide the actual halting status of the precondition. To properly capture the halting status, we define a top level statement construct like we discussed in the context of big-step and small-step SOS above, say `top` *Stmt*, modify the rule (MSOS-VAR) from

$$\texttt{int}\ xl\texttt{;}\ s \xrightarrow{\{\text{state}'=xl\mapsto 0,\ ...\}} s$$

to

$$\texttt{int}\ xl\texttt{;}\ s \xrightarrow{\{\text{state}'=xl\mapsto 0,\ \text{halting}'=\mathit{false},\ ...\}} \texttt{top}\ s$$

to mark the top level statement, and then finally include the following three rules:

$$\frac{s \xrightarrow{\{\text{halting}'=\mathit{false},\ ...\}} s'}{\texttt{top}\ s \xrightarrow{\{\text{halting}'=\mathit{false},\ ...\}} \texttt{top}\ s'} \qquad \text{(MSOS-TOP-NORMAL)}$$

$$\texttt{top}\ \{\} \rightarrow \{\} \qquad \text{(MSOS-TOP-SKIP)}$$

$$\frac{s \xrightarrow{\{\text{halting}'=\mathit{true},\ ...\}} s}{\texttt{top}\ s \xrightarrow{\{\text{halting}'=\mathit{false},\ ...\}} \{\}} \qquad \text{(MSOS-TOP-HALTING)}$$

The use of a `top` construct like above seems unavoidable if we want to achieve modularity. Indeed, we managed to avoid it in the small-step SOS definition of abrupt termination in Section 3.5.3 (paying one additional small-step to dissolve the halting configuration), because the halting configurations were explicitly, and thus non-modularly propagated through each of the language constructs, so the entire program reduced to a halting configuration whenever a division by zero or a "`halt;`" statement was encountered. Unfortunately, that same approach does not work with MSOS (unless we want to break its modularity, like in SOS), because the syntax is not mutilated when an abrupt termination occurs. The halting signal is captured by the label of the transition. However, the label does not tell us when we are at the top level in order to dissolve the halting status. Adding a new label to hold the depth of the derivation, or at least whether we are the top or not, would require one to (non-modularly) change it in each rule. The use of an additional `top` construct like we did above appears to be the best trade-off between modularity and elegance here.

Note that, although MSOS can be mechanically translated into SOS by associating to each MSOS attribute an SOS configuration component, the solution above to support abrupt termination modularly in MSOS is *not*

modular when applied in SOS via the translation. Indeed, adding a new attribute in the label means adding a new configuration component, which already breaks the modularity of SOS. In other words, the MSOS technique above cannot be manually used in SOS to obtain a modular definition of abrupt termination in SOS.

Dynamic Threads

The small-step SOS rules for spawning threads in Section 3.5.4 straightforwardly turn into MSOS rules:

$$\frac{s \to s'}{\texttt{spawn}\ s \to \texttt{spawn}\ s'} \qquad \text{(MSOS-Spawn-Arg)}$$

$$\texttt{spawn}\ \{\} \to \{\} \qquad \text{(MSOS-Spawn-Skip)}$$

$$\frac{s_2 \to s_2'}{(\texttt{spawn}\ s_1)\ s_2 \to (\texttt{spawn}\ s_1)\ s_2'} \qquad \text{(MSOS-Spawn-Wait)}$$

$$(s_1\ s_2)\ s_3 \equiv s_1\ (s_2\ s_3) \qquad \text{(MSOS-Seq-Assoc)}$$

$$\mathit{Stmt}\ ::=\ \mid\ \texttt{spawn}\ \mathit{Stmt}$$

Even though the MSOS rules above are conceptually identical to the original small-step SOS rules, they are more modular because, unlike the former, they carry over unchanged when the configuration needs to change. Note that the structural identity stating the associativity of sequential composition, called (MSOS-Seq-Assoc) above, is still necessary, and so is the syntactic extension of `spawn` to take statements instead of blocks.

Local Variables

Section 3.5.5 showed how blocks with local variables can be desugared into a uniform `let` construct, and also gave the small-step SOS rules defining the semantics of `let`. Those rules can be immediately adapted into the following MSOS rules:

$$\frac{a \to a'}{\texttt{let}\ x = a\ \texttt{in}\ s \to \texttt{let}\ x = a'\ \texttt{in}\ s} \qquad \text{(MSOS-Let-Exp)}$$

$$\frac{s \xrightarrow{\{\text{state}=\sigma[i/x],\ \text{state}'=\sigma',\ ...\}} s'}{\texttt{let}\ x = i\ \texttt{in}\ s \xrightarrow{\{\text{state}=\sigma,\ \text{state}'=\sigma'[\sigma(x)/x],\ ...\}} \texttt{let}\ x = \sigma'(x)\ \texttt{in}\ s'} \qquad \text{(MSOS-Let-Stmt)}$$

$$\texttt{let}\ x = i\ \texttt{in}\ \{\} \to \{\} \qquad \text{(MSOS-Let-Done)}$$

Like for the other features, the MSOS rules are more modular than their small-step SOS variants.

Since programs are now just ordinary (closed) expressions and they are now executed in the empty state, the rule (MSOS-Var), namely

$$\texttt{int}\ xl;\ s \xrightarrow{\{\text{state}'=xl\mapsto 0,\ ...\}} s$$

needs to change into a rule of the from

$$s \xrightarrow{\{\text{state}'=\cdot,\ ...\}} ?$$

Unfortunately, regardless of what we place instead of "?", such a rule will not work. That is because there is nothing to prevent it to apply to any statement at any step during the reduction. To enforce it to happen only at the top of the program and only at the beginning of the reduction, we can wrap the original program (which is a statement) into a one-element configuration-like term $\langle s \rangle$. Then the rule (MSOS-VAR) can be replaced with the following rule:

$$\langle s \rangle \xrightarrow{\{\text{state'}=\cdot,\ ...\}} s$$

Putting Them All Together

The modularity of MSOS makes it quite easy to put all the features discussed above together and thus define the MSOS of IMP++. Effectively, we have to do the following:

1. We add the three label attributes used for the semantics of the individual features above, namely the read-write input attribute and the two write-only attributes output and halting.

2. We add all the MSOS rules of all the features above *unchanged* (nice!), except for the rule (MSOS-VAR) for programs (which changed several times, anyway).

3. To initialize the label attributes, we add a pairing construct $\langle s, \omega \rangle$ like we did when we added the input/output extension of IMP, where s is a statement (programs are ordinary statements now) and ω is a buffer, and replace the rule (MSOS-VAR) in Figure 3.24 with the following:

$$\langle s, \omega \rangle \xrightarrow{\{\text{state'}=\mathit{xl}\mapsto 0,\ \text{input'}=\omega,\ \text{halting'}=\mathit{false},\ ...\}} \texttt{top}\ s$$

It is important to note that the MSOS rules of the individual IMP extensions can be very elegantly combined into one language (IMP++). The rule (MSOS-VAR) had to globally change in order to properly initialize the attribute values, but nothing had to be done in the MSOS rules of any of the features in order to put it together with the other MSOS rules of the other features.

Unfortunately, even though each individual feature has its intended semantics, the resulting IMP++ language does not. We still have the same semantic problems with regards to concurrency that we had in the context of small-step SOS in Section 3.5.6. For example, the concurrency of `spawn` statements is limited to the blocks in which they appear. For example, a statement of the form $(\texttt{let}\ x = i\ \texttt{in}\ \texttt{spawn}\ s_1)\ s_2$ does not allow s_2 to be evaluated concurrently with s_1. The statement s_1 has to evaluate completely and then the `let` statement dissolved, before any step in s_2 can be performed. To fix this problem, we would have to adopt a different solution, like the one proposed in Section 3.5.7 in the context of small-step SOS. Thanks to its modularity, MSOS would make such a solution easier to implement than small-step SOS. Particularly, one can use the label mechanism to pass a spawned thread and its execution environment all the way to the top modularly, without having to propagate it explicitly through language constructs.

3.6.3 MSOS in Rewrite Logic

Like big-step and small-step SOS, we can also associate a conditional rewrite rule to each MSOS rule and hereby obtain a rewrite logic theory that faithfully (i.e., step-for-step) captures the MSOS definition. There could be different ways to do this. One way to do it is to first desugar the MSOS definition into a step-for-step equivalent small-step SOS definition as discussed above, and then use the faithful embedding of small-step SOS into rewrite logic discussed in Section 3.3.3. The problem with this approach is that the resulting small-step SOS definition, and implicitly the resulting rewrite logic definition, lack the modularity of the

original MSOS definition. In other words, if one wanted to extend the MSOS definition with rules that would require global changes to its corresponding SOS definition (e.g., ones adding new semantic components into the label/configuration), then one would also need to manually incorporate all those global changes in the resulting rewrite logic definition.

We first show that any MSOS proof system, say MSOS, can be mechanically translated into a rewrite logic theory, say $\mathcal{R}_{\text{MSOS}}$, in such a way that two important aspects of the original MSOS definition are preserved: 1) the corresponding derivation relations are step-for-step equivalent, that is, MSOS $\vdash C \to C'$ if and only if $\mathcal{R}_{\text{MSOS}} \vdash \mathcal{R}_{C \to C'}$, where $\mathcal{R}_{C \to C'}$ is the corresponding syntactic translation of the MSOS sequent $C \to C'$ into a rewrite logic sequent; and 2) $\mathcal{R}_{\text{MSOS}}$ is as modular as MSOS. Second, we apply our generic translation technique to the MSOS formal system MSOS(IMP) defined in Section 3.6.1 and obtain a rewrite logic semantics of IMP that is step-for-step equivalent to and as modular as MSOS(IMP). The modularity of MSOS(IMP) and of $\mathcal{R}_{\text{MSOS(IMP)}}$ will pay off when we extend IMP in Section 3.5. Finally, we show how $\mathcal{R}_{\text{MSOS(IMP)}}$ can be seamlessly defined in Maude, yielding another interpreter for IMP (in addition to those corresponding to the big-step and small-step SOS definitions of IMP in Sections 3.2.3 and 3.3.3).

Computationally and Modularly Faithful Embedding of MSOS into Rewrite Logic

Our embedding of MSOS into rewrite logic is very similar to that of small-step SOS, with one important exception: the non-syntactic components of the configuration are all grouped into a *record*, which is a multiset of *attributes*, each attribute being a pair associating appropriate semantic information to a *field*. This allows us to use multiset *matching* (or matching modulo associativity, commutativity, and identity) in the corresponding rewrite rules to extract the needed semantic information from the record, thus achieving not only a computationally equivalent embedding of MSOS into rewrite logic, but also one with the same degree of modularity as MSOS.

Formally, let us assume an arbitrary MSOS formal proof system. Let *Attribute* be a fresh sort and let *Record* be the sort **Bag{***Attribute***}** (that means that we assume all the infrastructure needed to define records as comma-separated bags, or multisets, of attributes). For each field *Field* holding semantic contents *Contents* that appears unprimed or primed in any of the labels on any of the transitions in any of the rules of the MSOS proof system, let us assume an operation "*Field* = _ : *Contents* → *Attribute*" (the name of this postfix unary operation is "*Field* = _"). Finally, for each syntactic category *Syntax* used in any transition that appears anywhere in the MSOS proof system, let us define a configuration construct "$\langle \_, \_ \rangle$: *Syntax* × *Record* → *Configuration*". We are now ready to define our transformation of an MSOS rule into a rewrite logic rule:

1. Translate it into an SOS rule, as discussed right above Section 3.6.1; we could also go directly from MSOS rules to rewrite logic rules, but we would have to repeat most of the steps from MSOS to SOS that were already discussed;

2. Group all the semantic components in the resulting SOS configurations into a corresponding record, where each semantic component translates into a corresponding attribute using a corresponding label;

3. Replace those attributes whose semantic components are not used anywhere in the rule by a generic variable of sort *Record*;

4. Finally, use the technique in Section 3.3.3 to transform the resulting SOS-like rules into rewrite rules, tagging the left-hand-side configurations with the ∘ symbol.

Applying the steps above, the four MSOS rules discussed right above Section 3.6.1 (namely the ones for variable assignment, printing to the output, sequential composition of statements, and variable lookup) translate into the following rewrite logic rules:

$$\begin{aligned}
\circ\langle X = I\,\texttt{;}, (\mathsf{state} = \sigma,\ \rho)\rangle &\to \langle \{\}, (\mathsf{state} = \sigma[I/X],\ \rho)\rangle \\
\circ\langle \texttt{print(}I\texttt{)}\,\texttt{;}, (\mathsf{output} = \omega,\ \rho)\rangle &\to \langle \{\}, (\mathsf{output} = \omega : I,\ \rho)\rangle \\
\circ\langle S_1\ S_2, \rho\rangle &\to \langle S_1'\ S_2, \rho'\rangle \quad \textbf{if} \quad \circ\langle S_1, \rho\rangle \to \langle S_1', \rho'\rangle \\
\circ\langle X, (\mathsf{state} = \sigma,\ \rho)\rangle &\to \langle \sigma(X), (\mathsf{state} = \sigma,\ \rho)\rangle
\end{aligned}$$

We use the same mechanism as for small-step SOS to obtain the reflexive and transitive many-step closure of the MSOS one-step transition relation. This mechanism was discussed in detail in Section 3.3.3; it essentially consists of adding a configuration marker $\star$ together with a rule "$\star Cfg \to \star Cfg'$ **if** $\circ Cfg \to Cfg'$" iteratively applying the one-step relation.

Theorem 16. ***(Faithful embedding of MSOS into rewrite logic)*** *For any MSOS definition* MSOS*, and any appropriate configurations C and C', the following equivalences hold:*

$$\text{MSOS} \vdash C \to C' \iff \mathcal{R}_{\text{MSOS}} \vdash \circ\,\overline{C} \to^1 \overline{C'} \iff \mathcal{R}_{\text{MSOS}} \vdash \circ\,\overline{C} \to \overline{C'}$$

$$\text{MSOS} \vdash C \to^\star C' \iff \mathcal{R}_{\text{MSOS}} \vdash \star\,\overline{C} \to \star\,\overline{C'}$$

where $\mathcal{R}_{\text{MSOS}}$ is the rewrite logic semantic definition obtained from MSOS *by translating each rule in* MSOS *as above. (Recall from Section 2.5 that $\to^1$ is the one-step rewriting relation obtained by dropping the reflexivity and transitivity rules of rewrite logic. Also, as C and C' are parameter-free—parameters only appear in rules—, $\overline{C}$ and $\overline{C'}$ are ground terms.)*

Like for the previous embeddings of big-step and small-step SOS into rewrite logic, let us elaborate on the apparent differences between MSOS and $\mathcal{R}_{\text{MSOS}}$ from a user perspective. The most visible difference is the SOS-like style of writing the rules, namely using configurations instead of labels, which also led to the inheritance of the $\circ$ mechanism from the embedding of SOS into rewrite logic. Therefore, the equivalent rewrite logic definition is slightly more verbose than the original MSOS definition. On the other hand, it has the advantage that it is more direct than the MSOS definition, in that it eliminates all the notational conventions. Indeed, if we strip MSOS out of its notational conventions and go straight to its essence, we find that that essence is precisely its use of multiset matching to modularly access the semantic components of the configuration. MSOS chose to do this on the labels, using specialized conventions for read-only/write-only/read-write components, while our rewrite logic embedding of MSOS does it in the configurations, uniformly for all semantic components. Where precisely this matching takes place is, in our view, less relevant. What is relevant and brings MSOS its modularity is that multiset matching *does* happen. Therefore, similarly to the big-step and small-step SOS representations in rewrite logic, we conclude that the rewrite theory $\mathcal{R}_{\text{MSOS}}$ *is* MSOS, and *not* an encoding of it.

Like for the previous embeddings of big-step and small-step SOS into rewriting logic, unfortunately, $\mathcal{R}_{\text{MSOS}}$ (and implicitly MSOS) still lacks the main strengths of rewrite logic, namely context-insensitive and parallel application of rewrite rules. Indeed, the rules of $\mathcal{R}_{\text{MSOS}}$ can only apply at the top, sequentially, so these rewrite theories corresponding to the faithful embedding of MSOS follow a rather poor rewrite logic style. Like for the previous embeddings, this is not surprising though and does not question the quality of our embeddings. All it says is that MSOS was not meant to have the capabilities of rewrite logic with regards to context-insensitivity and parallelism; indeed, all MSOS attempts to achieve is to address the lack of modularity of SOS and we believe that it succeeded in doing so. Unfortunately, SOS has several other major problems, which are discussed in Section 3.5.

sorts:
Attribute, *Record*, *Configuration*, *ExtendedConfiguration*
subsorts and aliases:
Record = **Bag**{*Attribute*}
Configuration < *ExtendedConfiguration*
operations:
$\mathsf{state} = \_ : \mathit{State} \to \mathit{Attribute}$ // more fields can be added by need
$\langle \_, \_ \rangle : \mathit{AExp} \times \mathit{Record} \to \mathit{Configuration}$
$\langle \_, \_ \rangle : \mathit{BExp} \times \mathit{Record} \to \mathit{Configuration}$
$\langle \_, \_ \rangle : \mathit{Stmt} \times \mathit{Record} \to \mathit{Configuration}$
$\langle \_ \rangle : \mathit{Pgm} \to \mathit{Configuration}$
$\circ \_ : \mathit{Configuration} \to \mathit{ExtendedConfiguration}$ // reduce one step
$\star \_ : \mathit{Configuration} \to \mathit{ExtendedConfiguration}$ // reduce all steps
rule:
$\star \mathit{Cfg} \to \star \mathit{Cfg}'$ **if** $\circ \mathit{Cfg} \to \mathit{Cfg}'$ // where $\mathit{Cfg}, \mathit{Cfg}'$ are variables of sort *Configuration*

Figure 3.25: Configurations and infrastructure for the rewrite logic embedding of MSOS(IMP).

MSOS of IMP in Rewrite Logic

We here discuss the complete MSOS definition of IMP in rewrite logic, obtained by applying the faithful embedding technique discussed above to the MSOS definition of IMP in Section 3.6.1. Figure 3.25 gives an algebraic definition of configurations as well as needed additional record infrastructure; all the sorts, operations, and rules in Figure 3.25 were already discussed either above or in Section 3.3.3. Figure 3.26 gives the rules of the rewrite logic theory $\mathcal{R}_{\text{MSOS(IMP)}}$ that is obtained by applying the procedure above to the MSOS of IMP in Figures 3.23 and 3.24. Like before, we used the rewrite logic convention that variables start with upper-case letters; if they are greek letters, then we use a similar but larger symbol (e.g., Σ instead of σ for variables of sort *State*, or $\Rho$ instead of ρ for variables of sort *Record*). The following corollary of Theorem 16 establishes the faithfulness of the representation of the MSOS of IMP in rewrite logic:

Corollary 6. MSOS(IMP) $\vdash C \to C' \iff \mathcal{R}_{\text{MSOS(IMP)}} \vdash \circ\overline{C} \to \overline{C'}$.

Therefore, there is no perceivable computational difference between the IMP-specific proof system MSOS(IMP) and generic rewrite logic deduction using the IMP-specific rewrite rules in $\mathcal{R}_{\text{MSOS(IMP)}}$; the two are faithfully equivalent. Moreover, by the discussion following Theorem 16, $\mathcal{R}_{\text{MSOS(IMP)}}$ is also as modular as MSOS(IMP). This will be further emphasized in Section 3.5, where we will extend IMP with several features, some of which requiring more attributes.

★ Maude Definition of IMP MSOS

Figure 3.27 shows a straightforward Maude representation of the rewrite theory $\mathcal{R}_{\text{MSOS(IMP)}}$ in Figures 3.26 and 3.25. The Maude module `IMP-SEMANTICS-MSOS` in Figure 3.27 is executable, so Maude, through its rewriting capabilities, yields an MSOS interpreter for IMP the same way it yielded big-step and small-step SOS interpreters in Sections 3.2.3 and 3.3.3, respectively; for example, the Maude rewrite command

```
rewrite * < sumPgm > .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are also irrelevant, so they were replaced by "..."):

$$\circ\langle X, (\text{state} = \sigma, \rho)\rangle \to \langle \sigma(X), (\text{state} = \sigma, \rho)\rangle \quad \textbf{if} \quad \sigma(X) \neq \bot$$

$$\begin{aligned}
\circ\langle A_1 + A_2, \rho\rangle &\to \langle A_1' + A_2, \rho'\rangle \quad \textbf{if} \quad \circ\langle A_1, \rho\rangle \to \langle A_1', \rho'\rangle \\
\circ\langle A_1 + A_2, \rho\rangle &\to \langle A_1 + A_2', \rho'\rangle \quad \textbf{if} \quad \circ\langle A_2, \rho\rangle \to \langle A_2', \rho'\rangle \\
\circ\langle I_1 + I_2, \rho\rangle &\to \langle I_1 +_{Int} I_2, \rho\rangle
\end{aligned}$$

$$\begin{aligned}
\circ\langle A_1 \;/\; A_2, \rho\rangle &\to \langle A_1' \;/\; A_2, \rho'\rangle \quad \textbf{if} \quad \circ\langle A_1, \rho\rangle \to \langle A_1', \rho'\rangle \\
\circ\langle A_1 \;/\; A_2, \rho\rangle &\to \langle A_1 \;/\; A_2', \rho'\rangle \quad \textbf{if} \quad \circ\langle A_2, \rho\rangle \to \langle A_2', \rho'\rangle \\
\circ\langle I_1 \;/\; I_2, \rho\rangle &\to \langle I_1 \;/_{Int} I_2, \rho\rangle \quad \textbf{if} \quad I_2 \neq 0
\end{aligned}$$

$$\begin{aligned}
\circ\langle A_1 \mathrel{<=} A_2, \rho\rangle &\to \langle A_1' \mathrel{<=} A_2, \rho'\rangle \quad \textbf{if} \quad \circ\langle A_1, \rho\rangle \to \langle A_1', \rho'\rangle \\
\circ\langle I_1 \mathrel{<=} A_2, \rho\rangle &\to \langle I_1 \mathrel{<=} A_2', \rho'\rangle \quad \textbf{if} \quad \circ\langle A_2, \rho\rangle \to \langle A_2', \rho'\rangle \\
\circ\langle I_1 \mathrel{<=} I_2, \rho\rangle &\to \langle I_1 \leq_{Int} I_2, \rho\rangle
\end{aligned}$$

$$\begin{aligned}
\circ\langle \texttt{!}\ B, \rho\rangle &\to \langle \texttt{!}\ B', \rho'\rangle \quad \textbf{if} \quad \circ\langle B, \rho\rangle \to \langle B', \rho'\rangle \\
\circ\langle \texttt{! true}, \rho\rangle &\to \langle \texttt{false}, \rho\rangle \\
\circ\langle \texttt{! false}, \rho\rangle &\to \langle \texttt{true}, \rho\rangle
\end{aligned}$$

$$\begin{aligned}
\circ\langle B_1 \;\&\&\; B2, \rho\rangle &\to \langle B_1' \;\&\&\; B2, \rho'\rangle \quad \textbf{if} \quad \circ\langle B_1, \rho\rangle \to \langle B_1', \rho'\rangle \\
\circ\langle \texttt{false} \;\&\&\; B2, \rho\rangle &\to \langle \texttt{false}, \rho\rangle \\
\circ\langle \texttt{true} \;\&\&\; B2, \rho\rangle &\to \langle B_2, \rho\rangle
\end{aligned}$$

$$\circ\langle \{\ S\ \}, \rho\rangle \to \langle S, \rho\rangle$$

$$\begin{aligned}
\circ\langle X = A\,\texttt{;}, \rho\rangle &\to \langle X = A'\,\texttt{;}, \rho'\rangle \quad \textbf{if} \quad \circ\langle A, \rho\rangle \to \langle A', \rho'\rangle \\
\circ\langle X = I\,\texttt{;}, (\text{state} = \sigma, \rho)\rangle &\to \langle \{\}, (\text{state} = \sigma[I/X], \rho)\rangle \quad \textbf{if} \quad \sigma(X) \neq \bot
\end{aligned}$$

$$\begin{aligned}
\circ\langle S_1\ S_2, \rho\rangle &\to \langle S_1'\ S_2, \rho'\rangle \quad \textbf{if} \quad \circ\langle S_1, \rho\rangle \to \langle S_1', \rho'\rangle \\
\circ\langle \{\}\ S_2, \rho\rangle &\to \langle S_2, \rho\rangle
\end{aligned}$$

$$\begin{aligned}
\circ\langle \texttt{if (}B\texttt{)}\ S_1\ \texttt{else}\ S_2, \rho\rangle &\to \langle \texttt{if (}B'\texttt{)}\ S_1\ \texttt{else}\ S_2, \rho'\rangle \quad \textbf{if} \quad \circ\langle B, \rho\rangle \to \langle B', \rho'\rangle \\
\circ\langle \texttt{if (true)}\ S_1\ \texttt{else}\ S_2, \rho\rangle &\to \langle S_1, \rho\rangle \\
\circ\langle \texttt{if (false)}\ S_1\ \texttt{else}\ S_2, \rho\rangle &\to \langle S_2, \rho\rangle
\end{aligned}$$

$$\circ\langle \texttt{while (}B\texttt{)}\ S, \rho\rangle \to \langle \texttt{if (}B\texttt{)}\ \{\ S\ \texttt{while (}B\texttt{)}\ S\ \}\ \texttt{else}\ \{\}, \rho\rangle$$

$$\circ\langle \texttt{int}\ Xl\texttt{;}\ S\rangle \to \langle S, (\text{state} = Xl \mapsto 0)\rangle$$

Figure 3.26: $\mathcal{R}_{\text{MSOS(IMP)}}$: the complete MSOS of IMP in rewrite logic.

```
mod IMP-CONFIGURATIONS-MSOS is including IMP-SYNTAX + STATE .
  sorts Attribute Record Configuration ExtendedConfiguration .
  subsort Attribute < Record .
  subsort Configuration < ExtendedConfiguration .
  op empty : -> Record .
  op _,_ : Record Record -> Record [assoc comm id: empty] .
  op state`=_ : State -> Attribute .
  op <_,_> : AExp Record -> Configuration .
  op <_,_> : BExp Record -> Configuration .
  op <_,_> : Stmt Record -> Configuration .
  op <_> : Pgm -> Configuration .
  op o_ : Configuration -> ExtendedConfiguration [prec 80] .   --- one step
  op *_ : Configuration -> ExtendedConfiguration [prec 80] .   --- all steps
  var Cfg Cfg' : Configuration .
 crl * Cfg => * Cfg' if o Cfg => Cfg' .
endm

mod IMP-SEMANTICS-MSOS is including IMP-CONFIGURATIONS-MSOS .
  var X : Id .   var R R' : Record . var Sigma Sigma' : State .
  var I I1 I2 : Int .  var Xl : List{Id} .  var S S1 S1' S2 : Stmt .
  var A A' A1 A1' A2 A2' : AExp .  var B B' B1 B1' B2 B2' : BExp .

 crl o < X,(state = Sigma, R) > => < Sigma(X),(state = Sigma, R) >
  if Sigma(X) =/=Bool undefined .
 crl o < A1 + A2,R > => < A1' + A2,R' > if o < A1,R > => < A1',R' > .
 crl o < A1 + A2,R > => < A1 + A2',R' > if o < A2,R > => < A2',R' > .
  rl o < I1 + I2,R > => < I1 +Int I2,R > .
 crl o < A1 / A2,R > => < A1' / A2,R' > if o < A1,R > => < A1',R' > .
 crl o < A1 / A2,R > => < A1 / A2',R' > if o < A2,R > => < A2',R' > .
 crl o < I1 / I2,R > => < I1 /Int I2,R > if I2 =/=Bool 0 .
 crl o < A1 <= A2,R > => < A1' <= A2,R' > if o < A1,R > => < A1',R' > .
 crl o < I1 <= A2,R > => < I1 <= A2',R' > if o < A2,R > => < A2',R' > .
  rl o < I1 <= I2,R > => < I1 <=Int I2,R > .
 crl o < ! B,R > => < ! B',R' > if o < B,R > => < B',R' > .
  rl o < ! true,R > => < false,R > .
  rl o < ! false,R > => < true,R > .
 crl o < B1 && B2,R > => < B1' && B2,R' > if o < B1,R > => < B1',R' > .
  rl o < false && B2,R > => < false,R > .
  rl o < true && B2,R > => < B2,R > .
  rl o < {S}, R > => < S,R > .
 crl o < X = A ;,R > => < X = A' ;,R' > if o < A,R > => < A',R' > .
 crl o < X = I ;,(state = Sigma, R) > => < {},(state = Sigma[I / X], R) >
  if Sigma(X) =/=Bool undefined .
 crl o < S1 S2,R > => < S1' S2,R' > if o < S1,R > => < S1',R' > .
  rl o < {} S2,R > => < S2,R > .
 crl o < if (B) S1 else S2,R > => < if (B') S1 else S2,R' > if o < B,R > => < B',R'  > .
  rl o < if (true) S1 else S2,R > => < S1,R > .
  rl o < if (false) S1 else S2,R > => < S2,R > .
  rl o < while (B) S,R > => < if (B) {S while (B) S} else {},R > .
  rl o < int Xl ; S > => < S,(state = Xl |-> 0) > .
endm
```

Figure 3.27: The MSOS of IMP in Maude, including the definition of configurations.

```
rewrites: 7632 in ... cpu (... real) (... rewrites/second)
result ExtendedConfiguration: * < {},state = (n |-> 0 & s |-> 5050) >
```

Note that the rewrite command above took the same number of rewrite steps as the similar command executed on the small-step SOS of IMP in Maude discussed in Section 3.3.3, namely 7632. This is not unexpected, because matching is not counted as rewrite steps, no matter how complex it is.

Like for the big-step and small-step SOS definitions in Maude, one can also use any of the general-purpose tools provided by Maude on the MSOS definition above. For example, one can exhaustively search for all possible behaviors of a program using the `search` command:

```
search * < sumPgm > =>! Cfg:ExtendedConfiguration .
```

As expected, only one behavior will be discovered because our IMP language so far is deterministic. The same number of states as in the case of small-step SOS will be generated by this search command, 1709.

3.6.4 Notes

Modular Structural Operational Semantics (MSOS) was introduced in 1999 by Mosses [51] and since then mainly developed by himself and his collaborators (e.g., [52, 53, 54]). In this section we used the implicitly-modular variant of MSOS introduced in [54], which, as acknowledged by the authors of [54], was partly inspired from discussions with us[8] To be more precise, we used a slightly simplified version of implicitly-modular MSOS here. In MSOS in its full generality, one can also declare some transitions *unobservable*; to keep the presentation simpler, we here omitted all the observability aspects of MSOS.

The idea of our representation of MSOS into rewrite logic adopted in this section is taken over from Serbanuta *et al.* [74]. At our knowledge, Meseguer and Braga [44] give the first representation of MSOS into rewrite logic. The representation in [44] also led to the development of the Maude MSOS tool [16], which was the core of Braga's doctoral thesis. What is different in the representation of Meseguer and Braga in [44] from ours is that the former uses two different types of configuration wrappers, one for the left-hand-side of the transitions and one for the right-hand-side; this was already discussed in Section 3.3.4.

3.6.5 Exercises

Exercise 120. *Redo all the exercises in Section 3.3.5 but for the MSOS of* IMP *discussed in Section 3.6.1 instead of its small-step SOS in Section 3.3.2. Skip Exercises 68, 69 and 76, since the SOS proof system there drops the syntactic components in the RHS configurations in transitions, making it unsuitable for MSOS. For the MSOS variant of Exercise 70, just follow the same non-modular approach as in the case of SOS and not the modular MSOS approach discussed in Section 3.6.2 (Exercise 123 addresses the modular MSOS variant of abrupt termination).*

Exercise 121. *Same as Exercise 86, but for MSOS instead of small-step SOS: add variable increment to* IMP*, like in Section 3.6.2.*

Exercise 122. *Same as Exercise 90, but for MSOS instead of small-step SOS: add input/output to* IMP*, like in Section 3.6.2.*

Exercise 123. *Same as Exercise 95, but for MSOS instead of small-step SOS: add abrupt termination to* IMP*, like in Section 3.6.2.*

[8] In fact, drafts of this book preceding [54] had already dropped the implicit labels in MSOS rules, for notational simplicity.

Exercise 124. *Same as Exercise 104, but for MSOS instead of small-step SOS: add dynamic threads to* IMP*, like in Section 3.6.2.*

Exercise 125. *Same as Exercise 109, but for MSOS instead of small-step SOS: add local variables using* `let` *to* IMP*, like in Section 3.6.2.*

Exercise 126. *This exercise asks to define* IMP++ *in MSOS, in various ways. Specifically, redo Exercises 114, 115, 116, 117, and 118, but for the MSOS of* IMP++ *discussed in Section 3.6.2 instead of its small-step SOS in Section 3.5.6.*

3.7 Reduction Semantics with Evaluation Contexts

The small-step SOS/MSOS approaches discussed in Sections 3.3 and 3.6 define a language semantics as a proof system whose rules are mostly conditional. The conditions of such rules allow to implicitly capture the program execution context as a *proof context*. This shift of focus from the informal notion of execution context to the formal notion of proof context has a series of advantages and it was, in fact, the actual point of formalizing language semantics using SOS. However, as the complexity of programming languages increased, in particular with the adoption of control-intensive statements like call/cc (e.g., Scheme) that can arbitrarily change the execution context, the need for an explicit representation of the execution context as a first-class citizen in the language semantics also increased. Reduction semantics with evaluation contexts (RSEC) is a variant of small-step SOS where the evaluation context may appear explicit in the term being reduced.

In an RSEC language definition one starts by defining the syntax of *evaluation contexts*, or simply just *contexts*, which is typically done by means of a context-free grammar (CFG). A context is a program or a fragment of program with a *hole*, where the hole, which is written □, is a placeholder for where the next computational step can take place. If c is an evaluation context and e is some well-formed appropriate fragment (expression, statement, etc.), then $c[e]$ is the program or fragment obtained by replacing the hole of c by e. Reduction semantics with evaluation contexts relies on a tacitly assumed (but rather advanced) parsing mechanism that takes a program or a fragment p and decomposes it into a context c and a subprogram or fragment e, called a *redex*, such that $p = c[e]$. This decomposition process is called *splitting* (of p into c and e). The inverse process, composing a redex e and a context c into a program or fragment p, is called *plugging* (of e into c). These splitting/plugging operations are depicted in Figure 3.28.

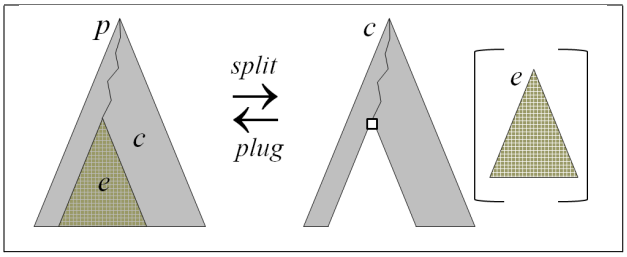

Figure 3.28: Decomposition of syntactic term p into context c and redex e, written $p = c[e]$: we say p *splits* into c and e, or e *plugs* into c yielding p. These operations are assumed whenever needed.

Consider a language with arithmetic/Boolean expressions and statements like our IMP language in Section 3.1. A possible CFG definition of evaluation contexts for such a language may include the following productions (the complete definition of IMP evaluation contexts is given in Figure 3.30):

Context ::= □
| *Context* <= *AExp*
| *Int* <= *Context*
| *Id* = *Context*;
| *Context Stmt*
| `if (`*Context*`)` *Stmt* `else` *Stmt*
| ...

Note how the intended evaluation strategies of the various language constructs are reflected in the definition of evaluation contexts: <= is sequentially strict (□ allowed to go into the first subexpression until evaluated to an *Int*, then into the second subexpression), the assignment is strict only in its second argument, while the sequential composition and the conditional are strict only in their first arguments. If one thinks of language constructs as operations taking a certain number of arguments of certain types, then note that the operations appearing in the grammar defining evaluation contexts are *different* from their corresponding operations

appearing in the language syntax; for example, "*Id* = *Context*;" is different from "*Id* = *AExp*;" because the former takes a context as second argument while the latter takes an arithmetic expression.

Here are some examples of correct evaluation contexts for the grammar above:

□

3 <= □

□ <= 3

□ x = 5;, where x is any variable.

`if (`□`)` s_1 `else` s_2, where s_1 and s_2 are any well-formed statements.

Here are some examples of incorrect evaluation contexts:

□ <= □ — a context can have only one hole.

x <= 3 — a context must contain a hole.

x <= □ — the first argument of <= must be an integer number in order to allow the hole in the second argument.

x = 5; □ — the hole can only appear in the first statement in a sequential composition.

□ = 5; — the hole cannot appear as first argument of the assignment construct.

`if (`x <= 7`)` □ `else` x = 5; — the hole is only allowed in the condition of a conditional.

Here are some examples of decompositions of syntactic terms into a context and a redex (recall that we can freely use parentheses for disambiguation; here we enclose evaluation contexts in parentheses for clarity):

$7 = (\square)[7]$

$3 <= x = (3 <= \square)[x] = (\square <= x)[3] = (\square)[3 <= x]$

$3 <= (2 + x) + 7 = (3 <= \square + 7)[2 + x] = (\square <= (2 + x) + 7)[3] = \ldots$

For simplicity, we consider only one type of context in this section, but in general one can have various types, depending upon the types of their holes and of their result.

Reduction semantics with evaluation contexts tends to be a purely syntactic definitional framework (following the slogan "everything is syntax"). If semantic components are necessary in a particular definition, then they are typically "swallowed by the syntax". For example, if one needs a state as part of the configuration for a particular language definition (like we need for our hypothetical IMP language discussed here), then one adds a context production of the form

$$\mathit{Context} \quad ::= \quad \langle \mathit{Context}, \mathit{State} \rangle$$

where the *State*, an inherently semantic entity, becomes part of the evaluation context. Note that once one adds additional syntax to evaluation contexts that does not correspond to constructs in the syntax of the original language, such as our pairing of a context and a state above, one needs to also extend the original syntax with corresponding constructs, so that the parsing-like mechanism decomposing a syntactic term into a context and a redex can be applied. In our case, the production above suggests that a pairing configuration construct of the form $\langle \mathit{Stmt}, \mathit{State} \rangle$, like for SOS, also needs to be defined. Unlike in SOS, we do not need configurations pairing other syntactic categories with a state, such as $\langle \mathit{AExp}, \mathit{State} \rangle$ and $\langle \mathit{BExp}, \mathit{State} \rangle$; the

reason is that, unlike in SOS, transitions with left-hand-side configurations $\langle Stmt, State\rangle$ are not derived anymore from transitions with left-hand-side configurations of the form $\langle AExp, State\rangle$ or $\langle BExp, State\rangle$.

Evaluation contexts are defined in such a way that whenever e is reducible, $c[e]$ is also reducible. For example, consider the term $c[i_1 \texttt{ <= } i_2]$ stating that expression $i_1 \texttt{ <= } i_2$ is in a proper evaluation context. Since $i_1 \texttt{ <= } i_2$ reduces to $i_1 \leq_{Int} i_2$, we can conclude that $c[i_1 \texttt{ <= } i_2]$ reduces to $c[i_1 \leq_{Int} i_2]$. Therefore, in reduction semantics with evaluation contexts we can define the semantics of <= using the following rule:

$$c[i_1 \texttt{ <= } i_2] \to c[i_1 \leq_{Int} i_2]$$

This rule is actually a rule schema, containing one rule instance for each concrete integers i_1, i_2 and for each appropriate evaluation context c. For example, here are instances of this rule when the context is $\square$, $\texttt{if (}\square\texttt{) \{\} else } x \texttt{ = 5;}$, and $\langle \square, (x \mapsto 1, y \mapsto 2)\rangle$, respectively:

$$\begin{array}{c} i_1 \texttt{ <= } i_2 \to i_1 \leq_{Int} i_2 \\ \texttt{if (}i_1 \texttt{ <= } i_2\texttt{) \{\} else } x \texttt{ = 5;} \to \texttt{if (}i_1 \texttt{ <= } i_2\texttt{) \{\} else } x \texttt{ = 5;} \\ \langle i_1 \texttt{ <= } i_2 \texttt{ , } (x \mapsto 1, y \mapsto 2)\rangle \to \langle i_1 \leq_{Int} i_2 \texttt{ , } (x \mapsto 1, y \mapsto 2)\rangle \end{array}$$

What is important to note here is that propagation rules, such as (MSOS-Leq-Arg1) and (MSOS-Leq-Arg2) in Figure 3.23, are not necessary anymore when using evaluation contexts, because the evaluation contexts already achieve the role of the propagation rules.

To reflect the fact that reductions take place only in appropriate contexts, RSEC typically introduces a rule schema of the form:

$$\frac{e \to e'}{c[e] \to c[e']} \qquad \text{(RSEC-Characteristic-Rule)}$$

where e, e' are well-formed fragments and c is any appropriate evaluation context (i.e., such that $c[e]$ and $c[e']$ are well-formed programs or fragments of program). This rule is called the *characteristic rule* of RSEC. When this rule is applied, we say that *e reduces to e' in context c*. If c is the empty context $\square$ then $c[e]$ is e and thus the characteristic rule is useless; for that reason, the characteristic rule may be encountered with a side condition "if $c \neq \square$". Choosing good strategies to search for splits of terms into contextual representations can be a key factor in obtaining efficient implementations of RSEC execution engines.

The introduction of the characteristic rule allows us to define reduction semantics of languages or calculi quite compactly. For example, here are all the rules needed to completely define the semantics of the comparison, sequential composition and conditional language constructs for which we defined evaluation contexts above:

$$\begin{array}{r} i_1 \texttt{ <= } i_2 \to i_1 \leq_{Int} i_2 \\ \texttt{\{\} } s_2 \to s_2 \\ \texttt{if (true) } s_1 \texttt{ else } s_2 \to s_1 \\ \texttt{if (false) } s_1 \texttt{ else } s_2 \to s_2 \end{array}$$

The characteristic rule tends to be the only conditional rule in an RSEC, in the sense that the remaining rules take no reduction premises (though they may still have side conditions). Moreover, as already pointed out, the characteristic rule is actually unnecessary, because one can very well replace each rule $l \to r$ by a rule $c[l] \to c[r]$. The essence of reduction semantics with evaluation contexts is not its characteristic reduction rule, but its specific approach to defining evaluation contexts as a grammar and then using them as an explicit part of languages or calculi definitions. The characteristic reduction rule can therefore be regarded as "syntactic sugar", or convenience to the designer allowing her to write more compact definitions.

To give the semantics of certain language constructs, one may need to access specific information that is stored inside an evaluation context. For example, consider a term $\langle x \texttt{ <= } 3,\ (x \mapsto 1, y \mapsto 2)\rangle$, which can be split as $c[x]$, where c is the context $\langle \square \texttt{ <= } 3,\ (x \mapsto 1, y \mapsto 2)\rangle$. In order to reduce $c[x]$ to $c[1]$ as desired, we need to look inside c and find out that the value of x in the state held by c is 1. Therefore, following the purely syntactic style adopted so far in this section, the reduction semantics with evaluation contexts rule for variable lookup in our case here is the following:

$$\langle c, \sigma\rangle[x] \rightarrow \langle c, \sigma\rangle[\sigma(x)] \quad \text{if } \sigma(x) \neq \bot$$

Indeed, the same way we add as much structure as needed in ordinary terms, we can add as much structure as needed in evaluation contexts. Similarly, below is the rule for variable assignment:

$$\langle c, \sigma\rangle[x \texttt{ = } i \texttt{;}] \rightarrow \langle c, \sigma[i/x]\rangle[\{\}] \quad \text{if } \sigma(x) \neq \bot$$

Note that in this case both the context and the redex were changed by the rule. In fact, as discussed in Section 3.10, one of the major benefits of reduction semantics with evaluation contexts consists in precisely the fact that one can arbitrarily modify the evaluation context in rules; this is crucial for giving semantics to control-intensive language constructs such as call/cc.

Splitting of a term into an evaluation context and a redex does not necessarily need to take place at the top of the left-hand-side of a rule. For example, the following is an alternative way to give reduction semantics with evaluation contexts to variable lookup and assignment:

$$\begin{array}{c} \langle c[x], \sigma\rangle \rightarrow \langle c[\sigma(x)], \sigma\rangle \quad \text{if } \sigma(x) \neq \bot \\ \langle c[x \texttt{ = } i \texttt{;}], \sigma\rangle \rightarrow \langle c[\{\}], \sigma[i/x]\rangle \quad \text{if } \sigma(x) \neq \bot \end{array}$$

Note that, even if we follow this alternative style, we still need to include the production *Context* ::= $\langle$*Context*, *State*$\rangle$ to the evaluation context CFG if we want to write rules as $c[i_1 \texttt{ <= } i_2] \rightarrow c[i_1 \leq_{Int} i_2]$ or to further take advantage of the characteristic rule and write elegant and compact rules such as $i_1 \texttt{ <= } i_2 \rightarrow i_1 \leq_{Int} i_2$. If we want to completely drop evaluation context productions that mix syntactic and semantic components, such as *Context* ::= $\langle$*Context*, *State*$\rangle$, then we may adopt one of the styles discussed in Exercises 127 and 128, respectively, though one should be aware of the fact that those styles also have their disadvantages.

Figure 3.29 shows a reduction sequence using the evaluation contexts and the rules discussed so far. We used the following (rather standard) notation for instantiated contexts whenever we applied the characteristic rule: the redex is placed in a box replacing the hole of the context. For example, the fact that expression $3 \texttt{ <= } x$ is split into contextual representation $(3 \texttt{ <= } \square)[x]$ is written compactly and intuitively as $3 \texttt{ <= } \boxed{x}$. Note that the evaluation context changes almost at each step during the reduction sequence in Figure 3.29.

Like in small-step SOS and MSOS, we can also transitively and reflexively close the one-step transition relation $\rightarrow$. As usual, we let $\rightarrow^\star$ denote the resulting multi-step transition relation.

3.7.1 The Reduction Semantics with Evaluation Contexts of IMP

Figure 3.30 shows the definition of evaluation contexts for IMP and Figure 3.31 shows all the reduction semantics rules of IMP using the evaluation contexts defined in Figure 3.30. The evaluation context productions capture the intended evaluation strategies of the various language constructs. For example, `+` and `/` are non-deterministically strict, so any one of their arguments can be reduced one step whenever the sum or the division expression can be reduced one step, respectively, so the hole $\square$ can go in any of their two subexpressions. As previously discussed, in the case of `<=` one can reduce its second argument only after its first argument is fully reduced (to an integer). The evaluation strategy of `!` is straightforward. For `&&`, note

$$
\begin{array}{ll}
 & \langle \boxed{x = 1;}\ y = 2;\ \texttt{if}\,(x <= y)\,\{\,x = 0;\,\}\,\texttt{else}\,\{\,y = 0;\,\},\ (x \mapsto 0, y \mapsto 0)\rangle \\
\rightarrow & \langle \boxed{\{\}\ y = 2;}\ \texttt{if}\,(x <= y)\,\{\,x = 0;\,\}\,\texttt{else}\,\{\,y = 0;\,\},\ (x \mapsto 1, y \mapsto 0)\rangle \\
\rightarrow & \langle \boxed{y = 2;}\ \texttt{if}\,(x <= y)\,\{\,x = 0;\,\}\,\texttt{else}\,\{\,y = 0;\,\},\ (x \mapsto 1, y \mapsto 0)\rangle \\
\rightarrow & \langle \{\}\ \texttt{if}\,(x <= y)\,\{\,x = 0;\,\}\,\texttt{else}\,\{\,y = 0;\,\},\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if}\,(\boxed{x} <= y)\,\{\,x = 0;\,\}\,\texttt{else}\,\{\,y = 0;\,\},\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if}\,(1 <= \boxed{y})\,\{\,x = 0;\,\}\,\texttt{else}\,\{\,y = 0;\,\},\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if}\,(\boxed{1 <= 2})\,\{\,x = 0;\,\}\,\texttt{else}\,\{\,y = 0;\,\},\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if}\,(\texttt{true})\,\{\,x = 0;\,\}\,\texttt{else}\,\{\,y = 0;\,\},\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle x = 0;,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \{\},\ (x \mapsto 0, y \mapsto 2)\rangle
\end{array}
$$

Figure 3.29: Sample reduction sequence.

IMP evaluation contexts syntax	IMP language syntax
Context ::= □ | *Context* + *AExp* | *AExp* + *Context* | *Context* / *AExp* | *AExp* / *Context* | *Context* <= *AExp* | *Int* <= *Context* | ! *Context* | *Context* && *BExp* | *Id* = *Context*; | *Context Stmt* | if (*Context*) *Stmt* else *Stmt*	*AExp* ::= *Int* | *Id* | *AExp* + *AExp* | *AExp* / *AExp* *BExp* ::= *Bool* | *AExp* <= *AExp* | ! *BExp* | *BExp* && *BExp* *Block* ::= {} | { *Stmt* } *Stmt* ::= *Block* | *Id* = *AExp*; | *Stmt Stmt* | if (*BExp*) *Block* else *Block* | while (*BExp*) *Block* *Pgm* ::= int **List**{*Id*}; *Stmt*

Figure 3.30: Evaluation contexts for IMP (left column); the syntax of IMP (from Figure 3.1) is recalled in the right column only for reader's convenience, to more easily compare the two grammars.

$$\mathit{Context} ::= \ldots \mid \langle \mathit{Context}, \mathit{State} \rangle$$

$$\frac{e \to e'}{c[e] \to c[e']}$$

$$\begin{aligned}
\langle c, \sigma \rangle[x] &\to \langle c, \sigma \rangle[\sigma(x)] \quad \text{if } \sigma(x) \neq \bot \\
i_1 + i_2 &\to i_1 +_{Int} i_2 \\
i_1 \texttt{ / } i_2 &\to i_1 /_{Int} i_2 \quad \text{if } i_2 \neq 0 \\
i_1 \texttt{ <= } i_2 &\to i_1 \leq_{Int} i_2 \\
\texttt{! true} &\to \texttt{false} \\
\texttt{! false} &\to \texttt{true} \\
\texttt{true \&\& } b_2 &\to b_2 \\
\texttt{false \&\& } b_2 &\to \texttt{false} \\
\texttt{\{ } s \texttt{ \}} &\to s \\
\langle c, \sigma \rangle[x \texttt{ = } i \texttt{;}] &\to \langle c, \sigma[i/x] \rangle[\texttt{\{\}}] \quad \text{if } \sigma(x) \neq \bot \\
\texttt{\{\}} \ s_2 &\to s_2 \\
\texttt{if (true) } s_1 \texttt{ else } s_2 &\to s_1 \\
\texttt{if (false) } s_1 \texttt{ else } s_2 &\to s_2 \\
\texttt{while (} b \texttt{) } s &\to \texttt{if (} b \texttt{) \{ } s \texttt{ while (} b \texttt{) } s \texttt{ \} else \{\}} \\
\langle \texttt{int } xl \texttt{; } s \rangle &\to \langle s, (xl \mapsto 0) \rangle
\end{aligned}$$

Figure 3.31: RSEC(IMP): The reduction semantics with evaluation contexts of IMP ($e, e' \in \mathit{AExp} \cup \mathit{BExp} \cup \mathit{Stmt}$; $c \in \mathit{Context}$ appropriate (that is, the respective terms involving c are well-formed); $i, i_1, i_2 \in \mathit{Int}$; $x \in \mathit{Id}$; $b, b_2 \in \mathit{BExp}$; $s, s_1, s_2 \in \mathit{Stmt}$; $xl \in \mathbf{List\{}\mathit{Id}\mathbf{\}}$; $\sigma \in \mathit{State}$).

that only its first argument is reduced. Indeed, recall that `&&` has a short-circuited semantics, so its second argument is reduced only after the first one is completely reduced (to a Boolean) and only if needed; this is defined using rules in Figure 3.31. The evaluation contexts for assignment, sequential composition, and the conditional have already been discussed.

Many of the rules in Figure 3.31 have already been discussed or are trivial. Note that there is no production *Context* ::= `while (`*Context*`)` *Stmt* as a hasty reader may (mistakenly) expect. That is because such a production would allow the evaluation of the Boolean expression in the while loop's condition to a Boolean value in the current context; supposing that value is `true`, then, unless one modifies the syntax in some rather awkward way, we cannot recover the original Boolean expression to evaluate it again after the evaluation of the while loop's body statement. The solution to handle loops remains the same as in SOS, namely to explicitly unroll them into conditional statements, as shown in Figure 3.31. Note that the evaluation contexts allow the loop unrolling to only happen when the `while` statement is a redex. In particular, after an unrolling reduction takes place, subsequent unrolling steps are disallowed inside the `then` branch; to unroll it again, the loop statement must become again a redex, which can only happen after the conditional statement is itself reduced. The initial program configuration (containing only the program) is reduced also like in SOS (last rule in Figure 3.31; note that one cannot instead define a production *Context* ::= `int` **List{***Id***};** *Context*, because, for example, there is no way to reduce "x = 5;" in $\langle$`int` x`;` $\boxed{x = 5;}\rangle$.

3.7.2 The Reduction Semantics with Evaluation Contexts of IMP++

We next discuss the reduction semantics of IMP++ using evaluation contexts. Like for the other semantics, we first add each feature separately to IMP and then we add all of them together and investigate the modularity and appropriateness of the resulting definition.

Variable Increment

The use of evaluation contexts makes the definition of variable increment quite elegant and modular:

$$\langle c, \sigma\rangle[{+}{+}\, x] \quad \rightarrow \quad \langle c, \sigma[(\sigma(x) +_{Int} 1)/x]\rangle[\sigma(x) +_{Int} 1]$$

No other rule needs to change, because: (1) unlike in SOS, all the language-specific rules are unconditional, each rule matching and modifying only its relevant part of the configuration; and (2) the language-independent characteristic rule allows reductions to match and modify only their relevant part of the configuration, propagating everything else in the configuration automatically.

Input/Output

We need to first change the configuration employed by our reduction semantics with evaluation contexts of IMP from $\langle s, \sigma\rangle$ to $\langle s, \sigma, \omega_{in}, \omega_{out}\rangle$, to also include the input/output buffers. This change, unfortunately, generates several other changes in the existing semantics, some of them non-modular in nature. First, we need to change the syntax of (statement) configurations to include input and output buffers, and the syntax of contexts from $\mathit{Context} ::= \ldots \mid \langle \mathit{Context}, \mathit{State}\rangle$ to $\mathit{Context} ::= \ldots \mid \langle \mathit{Context}, \mathit{State}, \mathit{Buffer}, \mathit{Buffer}\rangle$. No matter what semantic approach one employs, some changes in the configuration (or its equivalent) are unavoidable when one adds new language features that require new semantic data, like input/output constructs that require their own buffers (recall that, e.g., in MSOS, we had to add new attributes in transition labels instead). Hence, this change is acceptable. What is inconvenient (and non-modular), however, is that the rules for variable lookup and for assignment need the complete configuration, so they have to change from

$$\begin{array}{rcl} \langle c, \sigma\rangle[x] & \rightarrow & \langle c, \sigma\rangle[\sigma(x)] \\ \langle c, \sigma\rangle[x = i;] & \rightarrow & \langle c, \sigma[i/x]\rangle[\{\}] \end{array}$$

to

$$\begin{array}{rcl} \langle c, \sigma, \omega_{in}, \omega_{out}\rangle[x] & \rightarrow & \langle c, \sigma, \omega_{in}, \omega_{out}\rangle[\sigma(x)] \\ \langle c, \sigma, \omega_{in}, \omega_{out}\rangle[x = i;] & \rightarrow & \langle c, \sigma[i/x], \omega_{in}, \omega_{out}\rangle[\{\}] \end{array}$$

Also, the initial configuration which previously held only the program, now has to change to hold both the program and an input buffer, and the rule for programs (the last in Figure 3.31) needs to change as follows:

$$\langle \texttt{int}\, xl;\ s, \omega_{in}\rangle \rightarrow \langle s, (xl \mapsto 0), \omega_{in}, \epsilon\rangle$$

Once the changes above are applied, we are ready for adding the evaluation context for the print statement as well as the reduction semantics rules of both input/output constructs:

$$\begin{array}{rcl} \mathit{Context} & ::= & \ldots \mid \texttt{print(}\mathit{Context}\texttt{);} \\ \langle c, \sigma, i : \omega_{in}, \omega_{out}\rangle[\texttt{read()}] & \rightarrow & \langle c, \sigma, \omega_{in}, \omega_{out}\rangle[i] \\ \langle c, \sigma, \omega_{in}, \omega_{out}\rangle[\texttt{print(}i\texttt{);}] & \rightarrow & \langle c, \sigma, \omega_{in}, \omega_{out} : i\rangle[\{\}] \end{array}$$

Other possibilities to add input/output buffers to the configuration and to give the reduction semantics with evaluation contexts of the above language features in a way that appears to be more modular (but which yields other problems) are discussed in Section 3.10.

Abrupt Termination

For our language, reduction semantics allows very elegant, natural and modular definitions of abrupt termination, without having to extent the syntax of the original language and without adding any new reduction steps as an artifact of the approach chosen:

$$\begin{array}{rcl} \langle c, \sigma\rangle[i \ / \ o] & \rightarrow & \langle \{\}, \sigma\rangle \\ \langle c, \sigma\rangle[\texttt{halt;}] & \rightarrow & \langle \{\}, \sigma\rangle \end{array}$$

Therefore, the particular evaluation context in which the abrupt termination is being generated, c, is simply discarded. This is not possible in any of the big-step, small-step or MSOS styles above, because in there the evaluation context c is captured by the proof context, which, like in any logical system, cannot be simply discarded. The elegance of the two rules above suggests that having the possibility to explicitly match and change the evaluation context is a very powerful and convenient feature of a language semantic framework.

Dynamic Threads

The rules (SmallStep-Spawn-Arg) (resp. (MSOS-Spawn-Arg)) and (SmallStep-Spawn-Wait) (resp. (MSOS-Spawn-Wait)) in Section 3.5.4 (resp. Section 3.6.2) are essentially computation propagation rules. In reduction semantics with evaluation contexts the role of such rules is taken over by the splitting/plugging mechanism, which in turn relies on parsing and therefore needs productions for evaluation contexts. We can therefore replace those rules by appropriate productions for evaluation contexts:

$$\begin{array}{rcl} \textit{Context} & ::= & \ldots \\ & | & \texttt{spawn}\ \textit{Context} \\ & | & \texttt{spawn}\ \textit{Stmt}\ \textit{Context} \end{array}$$

The second evaluation context production above involves two language constructs, namely `spawn` and sequential composition. The desired non-determinism due to concurrency is captured by deliberate ambiguity in parsing evaluation contexts and, implicitly, in the splitting/plugging mechanism.

The remaining rule (SmallStep-Spawn-Skip) (resp. (MSOS-Spawn-Skip)) in Section 3.5.4 (resp. Section 3.6.2) is turned into an equivalent rule here, and the structural identity stating the associativity of sequential composition and the syntactic extension of `spawn` to take statements (instead of blocks) are also still necessary:

$$\begin{array}{rcl} \texttt{spawn}\ \{\} & \rightarrow & \{\} \\ (s_1\ s_2)\ s_3 & \equiv & s_1\ (s_2\ s_3) \\ \textit{Stmt} & ::= & \mid\ \texttt{spawn}\ \textit{Stmt} \end{array}$$

Local Variables

We make use of the procedure presented in Section 3.5.5 for desugaring blocks with local variables into `let` constructs, to reduce the problem to only give semantics to `let`. Recall from Section 3.5.5 that the semantics of $\texttt{let}\ x = a\ \texttt{in}\ s$ in a state σ is to first evaluate arithmetic expression a in σ to some integer i and then evaluate statement s in state $\sigma[i/x]$; after the evaluation of s, the value of x is recovered to $\sigma(x)$ (i.e., whatever it was before the execution of the block) but all the other state updates produced by the evaluation of s are kept. These suggest the following:

$$\textit{Context} \quad ::= \quad \ldots \ \mid\ \texttt{let}\ \textit{Id} = \textit{Context}\ \texttt{in}\ \textit{Stmt}$$

$$\langle c, \sigma\rangle[\texttt{let}\ x = i\ \texttt{in}\ s] \quad \rightarrow \quad \langle c, \sigma[i/x]\rangle[s\ x = \sigma(x)\texttt{;}]$$

Notice that a solution similar to that in small-step SOS and MSOS (see rules (SmallStep-Let-Stmt) and (MSOS-Let-Stmt) in Sections 3.5.5 and 3.6.2, respectively) does not work here, because rules in reduction semantics with evaluation contexts are unconditional. In fact, a solution similar to the one we adopted above was already discussed in Section 3.5.6 in the context of small-step SOS, where we also explained that it works as shown because of a syntactic trick, namely because we allow assignment statements of the form "$x = \bot$;" (in our case here when $\sigma(x) = \bot$), which have the effect to undefine x in σ (see Section 2.4.6). Section 3.5.6 also gives suggestions on how to avoid allowing $\bot$ to be assigned to x, if one does not like it. One suggestion was to rename the bound variable into a fresh one, this way relieving us from having to recover its value after the `let`:

$$\langle c, \sigma\rangle[\texttt{let}\ x = i\ \texttt{in}\ s] \rightarrow \langle c, \sigma[i/x']\rangle[s[x'/x]] \quad \text{if } x' \text{ is a fresh variable}$$

This approach comes with several problems, though: it requires that we define and maintain a substitution operation (for $s[x'/x]$), we have to pay a complexity linear with the size of the `let` body each time a `let` statement is eliminated, and the state may grow indefinitely (since the `let` can be inside a loop).

Note also that, with the current reduction semantics with evaluation contexts of IMP, we cannot add the following evaluation context production

$$\textit{Context} \quad ::= \quad \ldots \quad | \quad \texttt{let}\ \textit{Id} = \textit{Int}\ \texttt{in}\ \textit{Context}$$

stating that once the binding expression becomes an integer then we can evaluate the `let` body. We cannot add it simply because we have to bind the variable to the integer before we evaluate the `let` body statement. An evaluation context production like above would result in evaluating the `let` body statement in the same state as before the `let`, which is obviously wrong. However, the existence of a `let` binder allows us to possibly rethink the overall reduction semantics of IMP, to make it more syntactic. Indeed, the `let` binders can be used in a nested manner and hereby allow us to syntactically mimic a state. For example, a statement of the form `let` $x = 5$ `in let` $y = 7$ `in` s can be thought of as the statement s being executed in a "state" where x is bound to 5 and where y is bound to 7. We can then drop the configurations of IMP completely and instead add the evaluation context above allowing reductions inside `let` body statements. The IMP rules that refer to configurations, namely those for lookup and assignment, need to change to work with the new "state", and a new rule to eliminate unnecessary `let` statements needs to be added:

$$\begin{array}{rcl} \texttt{let}\ x = i\ \texttt{in}\ c[x] & \rightarrow & \texttt{let}\ x = i\ \texttt{in}\ c[i] \\ \texttt{let}\ x = i\ \texttt{in}\ c[x = j;] & \rightarrow & \texttt{let}\ x = j\ \texttt{in}\ c[\{\}] \\ \texttt{let}\ x = i\ \texttt{in}\ \{\} & \rightarrow & \{\} \end{array}$$

The above works correctly only if one ensures that the evaluation contexts c do not contain other `let` $x =$ _ `in` _ evaluation context constructs, with the same x variable name as in the rules (the underscores can be any integer and context, respectively). One can do this by statically renaming the bound variables to have different names at parse-time, or by employing a substitution operation to do it dynamically. Note that the static renaming approach requires extra-care as the language is extended, particularly if new `let` statements can be generated dynamically by other language constructs. Another approach to ensure the correct application of the rules above, which is theoretically more complex and practically more expensive and harder to implement, is to add a side condition to the first two rules of the form "where c does not contain any evaluation context production instance of the form `let` $x =$ _ `in` _".

To conclude, the discussion above suggests that there are various ways to give a reduction semantics of `let` using evaluation contexts, none of them absolutely better than the others: some are simpler, others

are more syntactic but require special external support, others require non-modular changes to the existing language. The approaches above are by no means exhaustive. For example, we have not even discussed environment-store based approaches.

Putting Them All Together

It is relatively easy to combine all the reduction semantics with evaluation contexts of all the features above, although not as modularly as it was for MSOS. Specifically, we have to do the following:

1. Apply all the changes that we applied when we added input/output to IMP above, namely: add input/output buffers to both configurations and configuration evaluation contexts; change the semantic rules involving configurations to work with the extended configurations, more precisely the rules for variable lookup, for variable assignment, and for programs.

2. Add all the evaluation contexts and the rules for the individual features above, making sure we *change* those of them using configurations or configuration evaluation contexts (i.e., almost all of them) to work with the new configurations including input/output buffers.

Unfortunately, the above is not giving us the desired language. Worse, it actually gives us a wrong language, namely one with a disastrous feature interaction. This problem has already been noted in Section 3.5.6, where we discussed the effect of a similar semantics to that of `let` above, but using small-step SOS instead of evaluation contexts. The problem here is that, if the body of a `let` statement contains a `spawn` statement, then the latter will be allowed, according to its semantics, to be executed in parallel with the statements following it. In our case, the assignment statement $x = \sigma(x)$`;` in the `let` semantics, originally intended to recover the value of x, can be now potentially executed before the spawned statement, resulting in a wrong behavior; in particular, the assignment can even "undefine" x in case $\sigma(x) = \bot$, in which case the `spawn` statement can even get stuck.

As already indicated in Section 3.5.6, the correct way to eliminate the `let` construct is to rename the bound variable into a fresh variable visible only to `let`'s body statement, this way eliminating the need to recover the value of the bound variable to what it was before the `let`:

$$\langle c, \sigma, \omega_{in}, \omega_{out}\rangle[\texttt{let } x = i \texttt{ in } s] \rightarrow \langle c, \sigma[i/x'], \omega_{in}, \omega_{out}\rangle[s[x'/x]] \quad \text{if } x' \text{ is a fresh variable}$$

We have used configurations already extended with input/output buffers, as needed for IMP++. This solution completely brakes any (intended or unintended) relationship between the `let` construct and any other language constructs that may be used inside its body, although, as discussed in Section 3.5.6, the use of the substitution comes with a few (relatively acceptable) drawbacks.

3.7.3 Reduction Semantics with Evaluation Contexts in Rewrite Logic

In this section we show how to automatically and faithfully embed reduction semantics with evaluation contexts into rewrite logic. After discussing how to embed evaluation contexts into rewrite logic, we first give a straightforward embedding of reduction semantics, which is easy to prove correct but which does not take advantage of performance-improving techniques currently supported by rewrite engines, so consequently it is relatively inefficient when executed or formally analyzed. We then discuss simple optimizations which increase the performance of the resulting rewrite definitions an order of magnitude or more. We only consider evaluation contexts which can be defined by means of context-free grammars (CFGs). However, the CFG that we allow for defining evaluation contexts can be non-deterministic, in the sense that a term is allowed to split many different ways into a context and a redex (like the CFG in Figure 3.30).

sort:
Syntax // includes all syntactic terms, in contextual representation *context*[*redex*] or not
subsorts:
$N_1, N_2, \ldots < Syntax$ // $N_1, N_2, \ldots$, are sorts whose terms can be regarded as *context*[*redex*]
operations:
$\_[\_] : Context \times Syntax \rightarrow Syntax$ // constructor for terms in contextual representation
$split : Syntax \rightarrow Syntax$ // puts syntactic terms into contextual representation
$plug : Syntax \rightarrow Syntax$ // the dual of split
rules and equations:
$split(Syn) \rightarrow \Box[Syn]$ // generic rule; it initiates the splitting process for the rules below
$plug(\Box[Syn]) = Syn$ // generic equation; it terminates the plugging process
// for each context production $Context ::= \pi(N_1, \ldots, N_n, Context)$ add the following:
$split(\pi(T_1, \ldots, T_n, T)) \rightarrow \pi(T_1, \ldots, T_n, C)[Syn]$ **if** $split(T) \rightarrow C[Syn]$
$plug(\pi(T_1, \ldots, T_n, C)[Syn]) = \pi(T_1, \ldots, T_n, plug(C[Syn]))$

Figure 3.32: Embedding evaluation contexts into rewrite logic theory $\mathcal{R}^{\Box}_{\text{RSEC}}$. The implicit split/plug mechanism is replaced by explicit rewrite logic sentences achieving the same task (the involved variables have the sorts $Syn : Syntax$, $C : Context$, $T_1 : N_1, \ldots, T_n : N_n$, and $T : N$).

Faithful Embedding of Evaluation Contexts into Rewrite Logic

Our approach to embedding reduction semantics with evaluation contexts in rewrite logic builds on an embedding of evaluation contexts and their implicit splitting/plugging mechanism in rewrite logic. More precisely, each evaluation context production is associated with an equation (for plugging) and a conditional rewrite rule (for splitting). The conditional rewrite rules allow to non-deterministically split a term into a context and a redex. Moreover, when executing the resulting rewrite logic theory, the conditional rules allow for finding *all* splits of a term into a context and a redex, provided that the underlying rewrite engine has search capabilities (like Maude does).

Figure 3.32 shows a general and automatic procedure to generate a rewrite logic theory from any CFG defining evaluation contexts for some given language syntax. Recall that, for simplicity, in this section we assume only one *Context* syntactic category. What links the CFG of evaluation contexts to the CFG of the language to be given a semantics, which is also what makes our embedding into rewrite logic discussed here work, is the assumption that for any context production

$$Context \quad ::= \quad \pi(N_1, \ldots, N_n, Context)$$

there are some syntactic categories N, N' (different or not) in the language CFG (possibly extended with configurations and semantic components as discussed above) such that $\pi(t_1, \ldots, t_n, t) \in N'$ for any $t_1 \in N_1$, $\ldots$, $t_n \in N_n$, $t \in N$. We here used a notation which needs to be explained. The actual production above is $Context ::= \pi$, where π is a string of terminals and non-terminals, but we write $\pi(N_1, \ldots, N_n, Context)$ instead of π to emphasize that $N_1, \ldots, N_n, Context$ are all the non-terminals appearing in π; we listed the *Context* last for simplicity. Also, by abuse of notation, we let $\pi(t_1, \ldots, t_n, t)$ denote the term obtained by substituting $(t_1, \ldots, t_n, t)$ for $(N_1, \ldots, N_n, Context)$ in π, respectively. So our assumption is that $\pi(t_1, \ldots, t_n, t)$ is well-formed under the syntax of the language whenever $t_1, \ldots, t_n, t$ are well-defined in the appropriate syntactic categories, that is, $t_1 \in N_1, \ldots, t_n \in N_n, t \in T$. This is indeed a very natural property of well-defined evaluation contexts for a given language, so natural that one may even ask how it can be otherwise (it is

easy to violate this property though, e.g., *Context* ::= <= *Context* = *AExp*;). Without this property, our embedding of evaluation contexts in rewrite logic in Figure 3.32 would not be well-formed, because the left-hand-side terms of some of the conditional rule(s) for split would not be well-formed terms.

For simplicity, in Figure 3.32 we prefer to subsort all the syntactic categories whose terms are intended to be allowed contextual representations *context*[*redex*] under one top sort[9], *Syntax*. The implicit notation *context*[*term*] for contextual representations, as well as the implicitly assumed *split* and *plug* operations, are defined explicitly in the corresponding rewrite theory. The split operation is only defined on terms over the original language syntax, while the plug operation is defined only over terms in contextual representation. One generic rule and one generic equation are added: $split(Syn) \rightarrow \Box[Syn]$ initiates the process of splitting a term into a contextual representation and $plug(\Box[Syn]) = Syn$ terminates the process of plugging a term into a context. It is important that the first be a rewrite rule (because it can lead to non-determinism; this is explained below), while the second can safely be an equation.

Each evaluation context production translates into one equation and one conditional rewrite rule. The equation tells how terms are plugged into contexts formed with that production, while the conditional rule tells how that production can be used to split a term into a context and a redex. The equations defining plugging are straightforward: for each production in the original CFG of evaluation contexts, iteratively plug the subterm in the smaller context; when the hole is reached, replace it by the subterm via the generic equation. The conditional rules for splitting also look straightforward, but how and why they work is more subtle. For any context production, if the term to split matches the pattern of the production, then first split the subterm corresponding to the position of the subcontext and then use that contextual representation of the subterm to construct the contextual representation of the original term; at any moment, one has the option to stop splitting thanks to the generic rule $split(Syn) \rightarrow \Box[Syn]$. For example, for the five *Context* productions in the evaluation context CFG in the preamble of this section, namely

Context ::= *Context* <= *AExp*
| *Int* <= *Context*
| *Id* = *Context*;
| *Context* *Stmt*
| `if (`*Context*`)` *Stmt* `else` *Stmt*

the general procedure in the rewrite logic embedding of evaluation contexts in Figure 3.32 yields the following five rules and five equations (variable I_1 has sort *Int*; *X* has sort *Id*; A_1 and A_2 have sort *AExp*; *B* has sort

[9] An alternative, which does not involve subsorting, is to rename all syntactic categories into one, *Syntax*. Our construction also works without subsorting and without collapsing of syntactic categories, but it is more technical, requires more operations, rules, and equations, and it is likely not worth the effort without a real motivation to use it in term rewrite settings without support for subsorting. We have made experiments with both approaches and found no penalty on performance when collapsing syntactic categories.

BExp; S_1 and S_2 have sort *Stmt*; C has sort *Context*; *Syn* has sort *Syntax*):

$$split(A_1 <= A_2) \to (C <= A_2)[Syn] \textbf{ if } split(A_1) \to C[Syn]$$
$$plug((C <= A_2)[Syn]) = plug(C[Syn]) <= A_2$$

$$split(I_1 <= A_2) \to (I_1 <= C)[Syn] \textbf{ if } split(A_2) \to C[Syn]$$
$$plug((I_1 <= C)[Syn]) = I_1 <= plug(C[Syn])$$

$$split(X = A\,;) \to (X = C\,;)[Syn] \textbf{ if } split(A) \to C[Syn]$$
$$plug((X = C\,;)[Syn]) = (X = plug(C[Syn])\,;)$$

$$split(S_1\ S_2) \to (C\ S_2)[Syn] \textbf{ if } split(S_1) \to C[Syn]$$
$$plug((C\ S_2)[Syn]) = plug(C[Syn])\ S_2$$

$$split(\texttt{if (}B\texttt{)}\ S_1\ \texttt{else}\ S_2) \to (\texttt{if (}C\texttt{)}\ S_1\ \texttt{else}\ S_2)[Syn] \textbf{ if } split(B) \to C[Syn]$$
$$plug((\texttt{if (}C\texttt{)}\ S_1\ \texttt{else}\ S_2)[Syn]) = \texttt{if (}plug(C[Syn])\texttt{)}\ S_1\ \texttt{else}\ S_2$$

The reason for which we used rules instead of equations for splitting in our embedding in Figure 3.32 is that splitting, unlike plugging, can be non-deterministic. Recall that the use of an arrow/transition in the condition of a rule has an existential nature (Sections 2.5). In particular, rewrite logic engines should execute conditional rules by performing an exhaustive search, or reachability analysis of all the zero-, one- or more-step rewrites of the condition left-hand-side term ($split(T)$ in our case) into the condition right-hand-side term ($C[Syn]$ in our case). Such an exhaustive search explores all possible splits, or parsings, of a term into a contextual representation.

Theorem 17. ***(Embedding splitting/plugging into rewrite logic)*** *Given an evaluation context CFG as discussed above, say as part of some reduction semantics with evaluation contexts definition* RSEC*, let* $\mathcal{R}^{\Box}_{\text{RSEC}}$ *be the rewrite logic theory associated to it as in Figure 3.32. Then the following are equivalent for any* $t, r \in$ *Syntax and* $c \in$ *Context:*

- t *can be split as* $c[r]$ *using the evaluation context CFG of* RSEC*;*
- $\mathcal{R}^{\Box}_{\text{RSEC}} \vdash split(t) \to c[r]$*;*
- $\mathcal{R}^{\Box}_{\text{RSEC}} \vdash plug(c[r]) = t$.

The theorem above says that the process of splitting a term t into a context and a redex in reduction semantics with evaluation contexts, which can be non-deterministic, reduces to reachability in the corresponding rewrite logic theory of a contextual representation pattern $c[r]$ of the original term marked for splitting, $split(t)$. Rewrite engines such as Maude provide a search command that does precisely that. We will shortly see how Maude's search command can find all splits of a term.

Faithful Embedding of RSEC in Rewriting Logic

In this section we discuss three faithful rewrite logic embeddings of reduction semantics with evaluation contexts. The first two assume that the embedded reduction semantics has no characteristic rule, in that all reductions take place at the top of the original term to reduce (e.g., a configuration in the case of our IMP language); this is not a limitation because, as already discussed, the characteristic rule can be regarded as syntactic sugar anyway, its role being to allow one to write reduction semantics definitions more compactly and elegantly. The first embedding is the simplest and easiest to prove correct, but it is the heaviest in notation

rules:
// for each reduction semantics rule $l(c_1[l_1], \ldots, c_n[l_n]) \to r(c'_1[r_1], \ldots, c'_{n'}[r_{n'}])$
// add the following conditional semantic rewrite rule:

$$\circ\, \overline{l}(T_1, \ldots, T_n) \to \overline{r}(plug(\overline{c'_1}[\overline{r_1}]), \ldots, plug(\overline{c'_n}[\overline{r_{n'}}])) \textbf{ if } split(T_1) \to \overline{c_1}[\overline{l_1}] \wedge \ \ldots \ \wedge split(T_n) \to \overline{c_n}[\overline{l_n}]$$

Figure 3.33: First embedding of RSEC into rewrite logic (RSEC $\rightsquigarrow \mathcal{R}^{\boxed{1}}_{\text{RSEC}}$).

and the resulting rewrite theories tend to be inefficient when executed because most of the left-hand-side terms of rules end up being identical, thus making the task of matching and selecting a rule to apply rather complex for rewrite engines. The second embedding results in rewrite rules whose left-hand-sides are mostly distinct, thus taking advantage of current strengths of rewrite engines to index terms so that rules to be applied can be searched for quickly. Our third embedding is as close to the original reduction semantics in form and shape as one can hope it to be in a rewriting setting; in particular, it also defines a characteristic rule, which can be used to write a more compact semantics. The third embedding yields rewrite theories which are as efficient as those produced by the second embedding. The reason we did not define directly the third embedding is because we believe that the transition from the first to the second and then to the third is instructive.

Since reduction semantics with evaluation contexts is an inherently small-step semantical approach, we use the same mechanism to control the rewriting as for small-step SOS (Section 3.3) and MSOS (Section 3.6). This mechanism was discussed in detail in Section 3.3.3. It essentially consists of: (1) tagging each left-hand-side term appearing in a rule transition with a $\circ$, to capture the desired notion of a one-step reduction of that term; and (2) tagging with a $\star$ the terms to be multi-step (zero, one or more steps) reduced, where $\star$ can be easily defined with a conditional rule as the transitive and reflexive closure of $\circ$ (see Section 3.3.3).

Figure 3.33 shows our first embedding of reduction semantics with evaluation contexts into rewrite logic, which assumes that the characteristic rule, if any, has already been desugared. Each reduction semantics rule translates into one conditional rewrite rule. We allow the reduction rules to have in their left-hand-side and right-hand-side terms an arbitrary number of subterms that are in contextual representation. For example, if the left-hand-side l of a reduction rule has n such subterms, say $c_1[l_1]$, ..., $c_n[l_n]$, then we write it $l(c_1[l_1], \ldots, c_n[l_n])$ (this is similar with our previous notation $\pi(N_1, \ldots, N_n, N)$ in the section above on embedding of evaluation contexts into rewrite logic, except that we now single out all the subterms in contextual representation instead of all the non-terminals). In particular, a rule $l \to r$ in which l and r contain no subterms in contextual representation (like the last rule in Figure 3.31) is translated exactly like in small-step SOS, that is, into $\overline{l} \to \overline{r}$. Also, note that we allow evaluation contexts to have any pattern (since we overline them, like any other terms); we do not restrict them to only be context variables. Consider, for example, the six reduction rules discussed in the preamble of Section 3.7, which after the desugaring of the characteristic rule are as follows:

$$\begin{aligned}
c[i_1 \texttt{ <= } i_2] &\to c[i_1 \leq_{Int} i_2] \\
c[\{\}\ s_2] &\to c[s_2] \\
c[\texttt{if (true) } s_1 \texttt{ else } s_2] &\to c[s_1] \\
c[\texttt{if (false) } s_1 \texttt{ else } s_2] &\to c[s_2] \\
\langle c, \sigma\rangle[x] &\to \langle c, \sigma\rangle[\sigma(x)] \quad \text{if } \sigma(x) \neq \bot \\
\langle c, \sigma\rangle[x = i;] &\to \langle c, \sigma[i/x]\rangle[\{\}] \quad \text{if } \sigma(x) \neq \bot
\end{aligned}$$

Since all these rules have left-hand-side terms already in contextual representation, their corresponding l in

Figure 3.33 is just a non-terminal (*Configuration*), which means that $\bar{l}$ is just a variable (of sort *Configuration*). Therefore, the rewrite logic rules associated to these RSEC rules are:

$$\circ\, Cfg \to plug(C[I_1 \leq_{Int} I_2])\ \textbf{if}\ split(Cfg) \to C[I_1 <= I_2]$$
$$\circ\, Cfg \to plug(C[S_2])\ \textbf{if}\ split(Cfg) \to C[\{\}\ S_2]$$
$$\circ\, Cfg \to plug(C[S_1])\ \textbf{if}\ split(Cfg) \to C[\texttt{if (true)}\ S_1\ \texttt{else}\ S_2]$$
$$\circ\, Cfg \to plug(C[S_2])\ \textbf{if}\ split(Cfg) \to C[\texttt{if (false)}\ S_1\ \texttt{else}\ S_2]$$
$$\circ\, Cfg \to plug(\langle C, \sigma\rangle[\sigma(X)])\ \textbf{if}\ split(Cfg) \to \langle C, \sigma\rangle[X]\ \wedge\ \sigma(X) \neq \bot$$
$$\circ\, Cfg \to plug(\langle C, \sigma[I/X]\rangle[\{\}])\ \textbf{if}\ split(Cfg) \to \langle C, \sigma\rangle[X = I\,;]\ \wedge\ \sigma(X) \neq \bot$$

Recall from the preamble of Section 3.7 that the RSEC rules for variable lookup and assignment can also be given as follows, so that their left-hand-side terms are not in contextual representation:

$$\langle c[x], \sigma\rangle \to \langle c[\sigma(x)], \sigma\rangle \quad \text{if } \sigma(x) \neq \bot$$
$$\langle c[x = i\,;], \sigma\rangle \to \langle c[\{\}], \sigma[i/x]\rangle \quad \text{if } \sigma(x) \neq \bot$$

In these cases, the string l in Figure 3.33 has the form $\langle Stmt, \sigma\rangle$, which means that $\bar{l}$ is the term $\langle S, \sigma\rangle$, where S is a variable of sort *Stmt*. Then their corresponding rewrite logic rules are:

$$\circ\, \langle S, \sigma\rangle \to \langle plug(C[\sigma(X)]), \sigma\rangle\ \textbf{if}\ split(S) \to C[X]\ \wedge\ \sigma(X) \neq \bot$$
$$\circ\, \langle S, \sigma\rangle \to \langle plug(C[\{\}]), \sigma[I/X]\rangle\ \textbf{if}\ split(S) \to C[X = I\,;]\ \wedge\ \sigma(X) \neq \bot$$

Once the characteristic rule is desugared as explained in the preamble of Section 3.7, an RSEC rule operates as follows: (1) attempt to match the left-hand-side pattern of the rule at the top of the term to reduce, making sure that each of the subterms corresponding to subpatterns in contextual representation form can indeed be split as indicated; and (2) if the matching step above succeeds, then reduce the original term to the right-hand-side pattern instantiated accordingly, plugging all the subterms appearing in contextual representations in the right-hand-side. Note that the conditional rewrite rule associated to an RSEC rule as indicated in Figure 3.33 achieves precisely the desired steps above: the $\circ$ in the left-hand-side term guarantees that the rewrite step takes place at the top of the original term, the condition exhaustively searches for the desired splits of the subterms in question into contextual representations, and the right-hand-side plugs back all the contextual representations into terms over the original syntax. The only difference between the original RSEC rule and its corresponding rewriting logic conditional rule is that the rewrite logic rule makes explicit the splits and plugs that are implicit in the RSEC rule.

Theorem 18. ***(First faithful embedding of reduction semantics into rewrite logic)*** *Let* RSEC *be any reduction semantics with evaluation contexts definition and let* $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ *be the rewrite logic theory associated to* RSEC *using the embedding procedures in Figures 3.32 and 3.33. Then*

1. ***(step-for-step correspondence)*** RSEC $\vdash t \to t'$ *using a reduction semantics with evaluation contexts rule iff* $\mathcal{R}^{\boxed{1}}_{\text{RSEC}} \vdash \circ\, \bar{t} \to^1 \overline{t'}$ *using the corresponding conditional rewrite rule obtained like in Figure 3.33; moreover, the reduction rule and the corresponding rewrite rule apply similarly (same contexts, same substitution; all modulo the correspondence in Theorem 17);*

2. ***(computational correspondence)*** RSEC $\vdash t \to^\star t'$ *iff* $\mathcal{R}^{\boxed{1}}_{\text{RSEC}} \vdash \star\, \bar{t} \to \star\, \overline{t'}$.

The first item in Theorem 18 says that the resulting rewriting logic theory captures faithfully the small-step reduction relation of the original reduction semantics with evaluation contexts definition. The faithfulness

of this embedding (i.e., there is precisely one top-level application of a rewrite rule that corresponds to an application of a reduction semantics rule), comes from the fact that the consistent use of the $\circ$ tag inhibits any other application of any other rule on the tagged term. Therefore, like in small-step SOS and MSOS, a small-step in a reduction semantics definition also reduces to reachability analysis in the corresponding rewrite theory; one can also use the search capability of a system like Maude to find all the next terms that a given term evaluates to (Maude provides the capability to search for the first n terms that match a given pattern using up to m rule applications, where n and m are user-provided parameters).

The step-for-step correspondence above is stronger (and better) than the strong bisimilarity of the two definitions; for example, if a reduction semantics rule in RSEC can be applied in two different ways on a term to reduce, then its corresponding rewrite rule in $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ can also be applied in two different ways on the tagged term. The second item in Theorem 18 says that the resulting rewrite theory can be used to perform any computation possible in the original RSEC, and vice versa (the step-for-step correspondence is guaranteed in combination with the first item). Therefore, there is absolutely no difference between computations using RSEC and computations using $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$, except for irrelevant syntactic conventions/notations. This strong correspondence between reductions in RSEC and rewrites in $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ tells that $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ *is precisely* RSEC*, not an encoding of it*. In other words, RSEC can be faithfully regarded as a methodological fragment of rewrite logic, same like big-step SOS, small-step SOS, and MSOS.

The discussion above implies that, from a theoretical perspective, the rewrite logic embedding of reduction semantics in Figure 3.33 is as good as one can hope. However, its simplicity comes at a price in performance, which unfortunately tends to be at its worst precisely in the most common cases. Consider, for example, the six rewrite rules used before Theorem 18 to exemplify the embedding in Figure 3.33 (consider the variant for lookup and assignment rules where the contextual representation in the left-hand-side appears at the top—first variant). They all have the form:

$$\circ\, \mathit{Cfg} \rightarrow \ldots \;\; \textbf{if} \;\; \mathit{split}(\mathit{Cfg}) \rightarrow \ldots$$

In fact, as seen in Figure 3.38, all the rewrite rules in the rewrite logic theory corresponding to the RSEC of IMP have the same form. The reason the left-hand-side terms of these rewrite rules are the same and lack any structure is because the contextual representations in the left-hand-side terms of the RSEC rules appear at the top, with no structure above them, which is the most common type of RSEC rule encountered.

To apply a conditional rewrite rule, a rewrite engine first matches the left-hand-side and then performs the (exhaustive) search in the condition. In other words, the structure of the left-hand-side acts as a cheap guard for the expensive search. Unfortunately, since the left-hand-side of the conditional rewrite rules above has no structure, it will always match. That means that the searches in the conditions of all the rewrite rules will be, in the worst case, executed one after another until a split is eventually found (if any). If one thinks in terms of implementing RSEC in general, then this is what a naive implementation would do. If one thinks in terms of executing term rewrite systems, then this fails to take advantage of some important performance-increasing advances in term rewriting, such as *indexing* [72, 73, 2]. In short, indexing techniques use the structure of the left-hand-sides to augment the term structure with information about which rule can potentially be applied at which places. This information is dynamically updated, as the term is rewritten. If the rules' left-hand-sides do not significantly overlap, it is generally assumed that it takes constant time to find a matching rewrite rule. This is similar in spirit to hashing, where the access time into a hash table is generally assumed to take constant time when there are no or few key collisions. Thinking intuitively in terms of hashing, from an indexing perspective a rewrite system with rules having the same left-hand-sides is as bad as a hash table in which all accesses are collisions.

Ideally, in an efficient implementation of RSEC one would like to adapt/modify indexing techniques,

rules:

// for each term l that appears as left-hand-side of a reduction rule
// $l(c_1[l_1], \ldots, c_n[l_n]) \rightarrow \ldots$ with $n > 0$, add the following
// conditional rewrite rule (there could be one l for many reduction rules):

$\circ\, \bar{l}(T_1, \ldots, T_n) \rightarrow T$ **if** $\circ\, \bar{l}(split(T_1), \ldots, split(T_n)) \rightarrow T$

// for each reduction semantics rule $l(c_1[l_1], \ldots, c_n[l_n]) \rightarrow r(c'_1[r_1], \ldots, c'_{n'}[r_{n'}])$
// add the following (unconditional) semantic rewrite rule:

$\circ\, \bar{l}(\overline{c_1}[\overline{l_1}], \ldots, \overline{c_n}[\overline{l_n}]) \rightarrow \overline{r}(plug(\overline{c'_1}[\overline{r_1}]), \ldots, plug(\overline{c'_n}[\overline{r_{n'}}]))$

Figure 3.34: Second embedding of RSEC into rewrite logic (RSEC $\rightsquigarrow \mathcal{R}^{\boxed{2}}_{\text{RSEC}}$).

which currently work for context-insensitive term rewriting, or to invent new techniques serving the same purpose. This seems highly non-trivial and tedious, though. An alternative is to device embedding transformations of RSEC into rewrite logic that take better or full advantage of existing, context-insensitive indexing. Without context-sensitive indexing or other bookkeeping mechanisms hardwired in the reduction engine, due to the inherent non-determinism in parsing/splitting syntax into contextual representations, in the worst case one needs to search the entire term to find a legal position where a reduction can take place. While there does not seem that we can do much to avoid such an exhaustive search in the worst case, note that our first embedding in Figure 3.33 initiates such a search in the condition of every rewrite rule: since in practice many/most of the rewrite rules generated by the procedure in Figure 3.33 end up having the same left-hand-side, the expensive search for appropriate splittings is potentially invoked many times. What we'd like to achieve is: (1) activate the expensive search for splitting only once; and (2) for each found split, quickly test which rule applies and apply it. Such a quick test as desired in (2) can be achieved for free on existing rewrite systems that use indexing, such as Maude, if one slightly modifies the embedding translation of RSEC into rewrite logic as shown in Figure 3.34.

The main idea is to keep the structure of the left-hand-side of the RSEC rules in the left-hand-side of the corresponding rewrite rules. This structure is crucial for indexing. To allow it, one needs to do the necessary splitting as a separate step. The first type of rewrite rules in Figure 3.34, one per term appearing as a left-hand-side in any of the conditional rules generated following the first embedding in Figure 3.33, enables the splitting process on the corresponding contextual representations in the left-hand-side of the original RSEC rule. We only define such rules for left-hand-side terms having at least one subterm in contextual representation, because if the left-hand-side l has no such terms then the rule would be $\circ\, \bar{l} \rightarrow T$ **if** $\circ\, \bar{l} \rightarrow T$, which is useless and does not terminate.

The second type of rules in Figure 3.34, one per RSEC rule, have almost the same left-hand-sides as the original RSEC rules; the only difference is the algebraic notation (as reflected by the overlining). Their right-hand-sides plug the context representations, so that they always yield terms which are well-formed over the original syntax (possibly extended with auxiliary syntax for semantics components—configurations, states, etc.). Consider, for example, the six RSEC rules discussed in the preamble of Section 3.7, whose translation into rewrite rules following our first embedding in Figure 3.33 was discussed right above Theorem 18. Let us first consider the variant for lookup and assignment rules where the contextual representation in the left-hand side appears at the top. Since in all these rules the contextual representation appears at the top of their

left-hand-side, which in terms of the first embedding in Figure 3.33 means that their corresponding rewrite rules (in the first embedding) had the form $\circ\, Cfg \to \ldots$ **if** $split(Cfg) \to \ldots$, we only need to add one rule of the first type in Figure 3.34 for them, namely (l is the identity pattern, i.e., $\bar{l}$ is a variable):

$$\circ\, Cfg \to Cfg' \ \textbf{if} \ \circ\, split(Cfg) \to Cfg'$$

With this, the six rewrite rules of the second type in Figure 3.34 corresponding to the six RSEC rules under discussion are the following:

$$\begin{aligned}
\circ\, C[I_1 \texttt{ <= } I_2] &\to plug(C[I_1 \leq_{Int} I_2]) \\
\circ\, C[\{\}\ S_2] &\to plug(C[S_2]) \\
\circ\, C[\texttt{if (true) } S_1 \texttt{ else } S_2] &\to plug(C[S_1]) \\
\circ\, C[\texttt{if (false) } S_1 \texttt{ else } S_2] &\to plug(C[S_2]) \\
\circ\, \langle C, \sigma\rangle[X] &\to plug(\langle C, \sigma\rangle[\sigma(X)]) \ \textbf{if} \ \sigma(X) \neq \bot \\
\circ\, \langle C, \sigma\rangle[X = I\,;] &\to plug(\langle C, \sigma[I/X]\rangle[\{\}]) \ \textbf{if} \ \sigma(X) \neq \bot
\end{aligned}$$

If one prefers the second variant for the reduction rules of lookup and assignment, namely

$$\begin{aligned}
\langle c[x], \sigma\rangle &\to \langle c[\sigma(x)], \sigma\rangle \quad \text{if } \sigma(x) \neq \bot \\
\langle c[x = i\,;], \sigma\rangle &\to \langle c[\{\}], \sigma[i/x]\rangle \quad \text{if } \sigma(x) \neq \bot
\end{aligned}$$

then, since the left-hand-side of these rules is a pattern of the form $\langle Stmt, \sigma\rangle$ which in algebraic form (overlined) becomes a term of the form $\langle S, \sigma\rangle$, we need to add one more rewrite rule of the first type in Figure 3.34, namely

$$\circ\, \langle S, \sigma\rangle \to Cfg' \ \textbf{if} \ \circ\, \langle split(S), \sigma\rangle \to Cfg',$$

and to replace the rewrite rules for lookup and assignment above with the following two rules:

$$\begin{aligned}
\circ\, \langle C[X], \sigma\rangle &\to \langle plug(C[\sigma(X)]), \sigma\rangle \ \textbf{if} \ \sigma(X) \neq \bot \\
\circ\, \langle C[X = I\,;], \sigma\rangle &\to \langle plug(C[\{\}]), \sigma[I/X]\rangle \ \textbf{if} \ \sigma(X) \neq \bot
\end{aligned}$$

Theorem 19. ***(Second faithful embedding of reduction semantics in rewrite logic)*** *Let* RSEC *be any reduction semantics with evaluation contexts definition and let* $\mathcal{R}^{\boxed{2}}_{\text{RSEC}}$ *be the rewrite logic theory associated to* RSEC *using the embedding procedures in Figures 3.32 and 3.34. Then*

1. ***(step-for-step correspondence)*** RSEC $\vdash t \to t'$ *using a reduction semantics with evaluation contexts rule iff* $\mathcal{R}^{\boxed{2}}_{\text{RSEC}} \vdash \circ\, \bar{t} \to^1 \overline{t'}$ *using the corresponding rewrite rules obtained like in Figure 3.34 (first a conditional rule of the first type whose left-hand-side matches t, then a rule of the second type which solves, in one rewrite step, the condition of the first rule); moreover, the reduction rule and the corresponding rewrite rules apply similarly (same contexts, same substitution; all modulo the correspondence in Theorem 17);*

2. ***(computational correspondence)*** RSEC $\vdash t \to^\star t'$ *iff* $\mathcal{R}^{\boxed{2}}_{\text{RSEC}} \vdash \star\, \bar{t} \to \star\, \overline{t'}$.

Theorem 19 tells us that we can use our second rewriting logic embedding transformation in Figure 3.34 to seamlessly execute RSEC definitions on context-insensitive rewrite engines, such as Maude. This was also the case for our first embedding (Figure 3.33 and its corresponding Theorem 18). However, as explained above, in our second embedding the left-hand-side terms of the rewrite rules corresponding to the actual reduction semantics rules (the second type of rule in Figure 3.34) preserve the structure of the left-hand-side

rules:

// for each term l that appears as the left-hand-side of a reduction rule
// $l(c_1[l_1], \ldots, c_n[l_n]) \rightarrow \ldots$, add the following conditional
// rewrite rule (there could be one l for many reduction rules):

$$\circ\, \overline{l}(T_1, \ldots, T_n) \rightarrow T \ \textbf{if}\ plug(\circ\, \overline{l}(split(T_1), \ldots, split(T_n))) \rightarrow T$$

// for each non-identity term r appearing as right-hand-side in a reduction rule
// $\ldots \rightarrow r(c_1[r_1], \ldots, c_n[r_n])$, add the following equation
// (there could be one r for many reduction rules):

$$plug(\overline{r}(Syn_1, \ldots, Syn_n)) = \overline{r}(plug(Syn_1), \ldots, plug(Syn_n))$$

// for each reduction semantics rule $l(c_1[l_1], \ldots, c_n[l_n]) \rightarrow r(c'_1[r_1], \ldots, c'_{n'}[r_{n'}])$
// add the following semantic rewrite rule:

$$\circ\, \overline{l}(\overline{c_1}[\overline{l_1}], \ldots, \overline{c_n}[\overline{l_n}]) \rightarrow \overline{r}(\overline{c'_1}[\overline{r_1}], \ldots, \overline{c'_n}[\overline{r_{n'}}])$$

Figure 3.35: Third embedding of RSEC in rewrite logic (RSEC $\rightsquigarrow \mathcal{R}^{\boxed{3}}_{\text{RSEC}}$).

terms of the original corresponding reduction rules. This important fact has two benefits. On the one hand, the underlying rewrite engines can use that structure to enhance the efficiency of rewriting by means of indexing, as already discussed above. On the other hand, the resulting rewrite rules resemble the original reduction rules, so the language designer who wants to use our embedding feels more comfortable. Indeed, since the algebraic representation of terms (the overline) should not change the way they are perceived by a user, the only difference between the left-hand-side of the original reduction rule and the left-hand-side of the resulting rewrite rule is the $\circ$ symbol: $l(c_1[l_1], \ldots, c_n[l_n])$ versus $\circ\, \overline{l}(\overline{c_1}[\overline{l_1}], \ldots, \overline{c_n}[\overline{l_n}])$, e.g., $\langle c[x = i\,;], \sigma \rangle$ versus $\circ \langle C[X = I\,;], \sigma \rangle$, where c, σ, x, i are reduction rule parameters while C, σ, X, I are corresponding variables of appropriate sorts.

Even though the representational distance between the left-hand-side terms in the original reduction rules and the left-hand-side terms in the resulting rewrite rules is minimal (one cannot eliminate the $\circ$, as extensively discussed in Section 3.3.3), unfortunately, the same does not hold true for the right-hand-side terms. Indeed, a right-hand-side $r(c'_1[r_1], \ldots, c'_n[r_{n'}])$ of a reduction rule becomes the right-hand-side $\overline{r}(plug(\overline{c'_1}[\overline{r_1}]), \ldots, plug(\overline{c'_n}[\overline{r_{n'}}]))$ of its corresponding rewrite rule, e.g., $\langle c[\{\}], \sigma \rangle$ becomes $\langle plug(C[\{\}]), \sigma \rangle$.

Figure 3.35 shows our third and final embedding of RSEC in rewrite logic, which has the advantage that it completely isolates the uses of *split*/*plug* from the semantic rewrite rules. Indeed, the rewrite rule associated to a reduction rule has the same left-hand-side as in the second embedding, but now the right-hand-side is actually the algebraic variant of the right-hand-side of the original reduction rule. This is possible because of two simple adjustments of the second embedding:

1. To avoid having to explicitly use the *plug* operation in the semantic rewrite rules, we replace the first type of conditional rewrite rules in the second embedding, namely

$$\circ\, \overline{l}(T_1, \ldots, T_n) \rightarrow T \ \textbf{if}\ \ \circ\, \overline{l}(split(T_1), \ldots, split(T_n)) \rightarrow T,$$

with slightly modified conditional rewrite rules of the form

$$\circ\, \overline{l}(T_1, \ldots, T_n) \to T \ \textbf{if}\ plug(\circ\, \overline{l}(split(T_1), \ldots, split(T_n))) \to T.$$

Therefore, the left-hand-side term of the condition is wrapped with the *plug* operation. Since rewriting is context-insensitive, the *plug* wrapper does not affect the rewrites that happen underneath in the $\circ\, \overline{l}(\ldots)$ term. Like in the second embedding, the only way for $\circ$ to disappear from the condition left-hand-side is for a semantic rule to apply. When that happens, the left-hand-side of the condition is rewritten to a term of the form $plug(t)$, where t matches the right-hand-side of some reduction semantics rule, which may potentially contain some subterms in contextual representation.

2. To automatically plug all the subterms in contextual representation that appear in t after the left-hand-side term of the condition in the rule above rewrites to $plug(t)$, we add equations of the form

$$plug(\overline{r}(Syn_1, \ldots, Syn_n)) = \overline{r}(plug(Syn_1), \ldots, plug(Syn_n)),$$

one for each non-identity pattern r appearing as a right-hand side of an RSEC rule; if r is an identity pattern then the equation becomes $plug(Syn) = plug(Syn)$, so we omit it.

Let us exemplify our third rewrite logic embedding transformation of reduction semantics with evaluation contexts using the same six reduction rules used so far in this section, but, to make it more interesting, considering the second variant of reduction rules for variable lookup and assignment. We have two left-hand-side patterns in these reduction rules, namely *Configuration* and $\langle Stmt, \sigma \rangle$, so we have the following two rules of the first type in Figure 3.35:

$$\begin{array}{r} \circ\, Cfg \to Cfg' \ \textbf{if}\ plug(\circ\, split(Cfg)) \to Cfg' \\ \circ\, \langle S, \sigma \rangle \to Cfg' \ \textbf{if}\ plug(\circ\, \langle split(S), \sigma \rangle) \to Cfg' \end{array}$$

We also have two right-hand-side patterns in these reduction rules, the same two as above, but the first one is an identity pattern so we only add one equation of the second type in Figure 3.35:

$$plug(\langle C[Syn], \sigma \rangle) = \langle plug(C[Syn]), \sigma \rangle$$

We can now give the six rewrite rules corresponding to the six reduction rules in discussion:

$$\begin{array}{r} \circ\, C[I_1 \texttt{ <= } I_2] \to C[I_1 \leq_{Int} I_2] \\ \circ\, C[\{\}\ S_2] \to C[S_2] \\ \circ\, C[\texttt{if (true) } S_1 \texttt{ else } S_2] \to C[S_1] \\ \circ\, C[\texttt{if (false) } S_1 \texttt{ else } S_2] \to C[S_2] \\ \circ\, \langle C[X], \sigma \rangle \to \langle C[\sigma(X)], \sigma \rangle \ \textbf{if}\ \sigma(X) \neq \bot \\ \circ\, \langle C[X = I\texttt{;}], \sigma \rangle \to \langle C[\{\}], \sigma[I/X] \rangle \ \textbf{if}\ \sigma(X) \neq \bot \end{array}$$

The six rewrite rules above are as close to the original reduction semantics rules as one can hope them to be in a rewriting setting. Note that, for simplicity, we preferred to desugar the characteristic rule of reduction semantics with evaluation contexts in all our examples in this subsection. At this moment we have all the infrastructure needed to also include a rewrite equivalent of it:

$$\circ\, C[Syn] \to C[Syn'] \ \textbf{if}\ C \neq \Box \wedge \circ\, Syn \to Syn'$$

Note that we first check whether the context is proper in the condition of the characteristic rewrite rule above, and then we initiate a (small-step) reduction of the redex (by tagging it with the symbol $\circ$). The condition

is well-defined in rewrite logic because, as explained in Figure 3.32, we subsorted all the syntactic sorts together with the configuration under the top sort *Syntax*, so all these sorts belong to the same kind (see Section 2.5), which means that the operation $\circ$ can apply to any of them, including to *Syntax*, despite the fact that it was declared to take a *Configuration* to an *ExtendedConfiguration* (like in Section 3.3.3). With this characteristic rewrite rule, we can now restate the six rewrite rules corresponding to the six reduction rules above as follows:

$$\begin{aligned}
\circ\, I_1 <= I_2 &\to I_1 \leq_{Int} I_2\\
\circ\, \{\}\ S_2 &\to S_2\\
\circ\, \texttt{if (true)}\ S_1\ \texttt{else}\ S_2 &\to S_1\\
\circ\, \texttt{if (false)}\ S_1\ \texttt{else}\ S_2 &\to S_2\\
\circ\, \langle C[X], \sigma\rangle &\to \langle C[\sigma(X)], \sigma\rangle \ \textbf{if}\ \sigma(X) \neq \bot\\
\circ\, \langle C[X = I\,;], \sigma\rangle &\to \langle C[\{\}], \sigma[I/X]\rangle \ \textbf{if}\ \sigma(X) \neq \bot
\end{aligned}$$

Note, again, that $\circ$ is applied on arguments of various sorts in the same kind with *Configuration*.

The need for $\circ$ in the left-hand-side terms of rules like above is now even more imperative than before. In addition to all the reasons discussed so far, there are additional reasons now for which the dropping of $\circ$ would depart us from the intended faithful capturing of reduction semantics in rewrite logic. Indeed, if we drop $\circ$ then there is nothing to stop the applications of rewrite rules at any places in the term to rewrite, potentially including places which are not allowed to be evaluated yet, such as, for example, in the branches of a conditional. Moreover, such applications of rules could happen concurrently, which is strictly disallowed by reduction semantics with or without evaluation contexts. The role of $\circ$ is precisely to inhibit the otherwise unrestricted potential to apply rewrite rules everywhere and concurrently: rules are now applied sequentially and only at the top of the original term, exactly like in reduction semantics.

Theorem 20. ***(Third faithful embedding of reduction semantics into rewrite logic)*** *Let* RSEC *be any reduction semantics with evaluation contexts definition (with or without a characteristic reduction rule) and let* $\mathcal{R}^{\boxed{3}}_{\text{RSEC}}$ *be the rewrite logic theory associated to* RSEC *using the embedding procedures in Figures 3.32 and 3.35 (plus the characteristic rewrite rule above in case* RSEC *comes with a characteristic reduction rule). Then*

1. ***(step-for-step correspondence)*** RSEC $\vdash t \to t'$ *using a reduction semantics with evaluation contexts rule iff* $\mathcal{R}^{\boxed{3}}_{\text{RSEC}} \vdash \circ\, \bar{t} \to^1 \overline{t'}$;
2. ***(computational correspondence)*** RSEC $\vdash t \to^\star t'$ *iff* $\mathcal{R}^{\boxed{3}}_{\text{RSEC}} \vdash \star\, \bar{t} \to \star\, \overline{t'}$.

We can therefore safely conclude that RSEC has been captured as a methodological fragment of rewrite logic. The faithful embeddings of reduction semantics into rewrite logic above can be used in at least two different ways. On the one hand, they can be used as compilation steps transforming a context-sensitive reduction system into an equivalent context-insensitive rewrite system, which can be further executed/compiled/analyzed using conventional rewrite techniques and existing rewrite engines. On the other hand, the embeddings above are so simple, that one can simply use them manually and thus "think reduction semantics" in rewrite logic.

Reduction Semantics with Evaluation Contexts of IMP in Rewrite Logic

We here discuss the complete reduction semantics with evaluation contexts definition of IMP in rewrite logic, obtained by applying the faithful embedding techniques discussed above to the reduction semantics definition of IMP in Figure 3.31 in Section 3.7.1. We start by defining the needed configurations, then we give all the

sorts:
Configuration, *ExtendedConfiguration*
subsort:
Configuration < *ExtendedConfiguration*
operations:
$\langle \_, \_ \rangle$: *Stmt* × *State* → *Configuration*
$\langle \_ \rangle$: *Pgm* → *Configuration*
$\circ \_$: *Configuration* → *ExtendedConfiguration* // reduce one step
$\star \_$: *Configuration* → *ExtendedConfiguration* // reduce all steps
rule:
$\star Cfg \rightarrow \star Cfg'$ **if** $\circ Cfg \rightarrow Cfg'$ // where Cfg, Cfg' are variables of sort *Configuration*

Figure 3.36: Configurations and infrastructure for the rewrite logic embedding of RSEC(IMP).

rewrite rules and equations embedding the evaluation contexts and their splitting/plugging mechanism in rewrite logic, and then we finally give three rewrite theories corresponding to the three embeddings discussed above, each including the (same) configurations definition and embedding of evaluation contexts.

Figure 3.36 gives an algebraic definition of IMP configurations as needed for reduction semantics with evaluation contexts, together with the additional infrastructure needed to represent the one-step and multi-step transition relations. Everything defined in Figure 3.36 has already been discussed in the context of small-step SOS (see Figures 3.13 and 3.17 in Section 3.3.3). Note, however, that we only defined a subset of the configurations needed for small-step SOS, more precisely only the top-level configurations (ones holding a program and ones holding a statement and a state). The intermediate configurations holding expressions and a state in small-step SOS are not needed here because reduction semantics with evaluation contexts does not need to explicitly decompose bigger reduction tasks into smaller ones until a redex is eventually found, like small-step SOS does; instead, the redex is found atomically by splitting the top level configuration into a context and the redex.

Figure 3.37 shows the rewrite logic theory $\mathcal{R}^{\square}_{\text{RSEC(IMP)}}$ associated to the evaluation contexts of IMP in RSEC(IMP) (Figure 3.30) following the procedure described in Section 3.7.3 and summarized in Figure 3.32. Recall that all language syntactic categories and configurations are sunk into a top sort *Syntax*, and that one rule for splitting and one equation for plugging are generated for each context production. In general, the embedding of evaluation contexts tends to be the largest and the most boring portion of the rewrite logic embedding of a reduction semantics language definition. However, fortunately, this can be generated fully automatically. An implementation of the rewrite logic embedding techniques discussed in this section may even completely hide this portion from the user. We show it in Figure 3.37 only for the sake of completeness.

Figure 3.38 shows the rewrite logic theory $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ corresponding to the rules in the reduction semantics with evaluation contexts of IMP in Section 3.7.1, following our first embedding transformation depicted in Figure 3.33. Like before, we used the rewrite logic convention that variables start with upper-case letters; if they are Greek letters, then we use a similar but larger symbol (e.g., Σ instead of σ for variables of sort *State*). These rules are added, of course, to those corresponding to evaluation contexts in Figure 3.37 (which are common to all three embeddings). Note that there is precisely one conditional rewrite rule in Figure 3.38 corresponding to each reduction semantics rule of IMP in Figure 3.31. Also, note that if a rule does not make use of evaluation contexts, then its corresponding rewrite rule is identical to the rewrite rule corresponding to the small-step SOS embedding discussed in Section 3.3.3. For example, the last reduction rule in Figure 3.31 results in the last rewrite rule in Figure 3.38, which is identical to the last rewrite rule

sorts:

$Syntax,\ Context$

subsorts:

$AExp, BExp, Stmt, Configuration < Syntax$

operations:

$\square : \rightarrow Context$ $\qquad$ $\_[\_] : Context \times Syntax \rightarrow Syntax$

$split : Syntax \rightarrow Syntax$ $\qquad$ $plug : Syntax \rightarrow Syntax$

$\langle \_, \_ \rangle : Context \times State \rightarrow Context$

$\_ + \_ : Context \times AExp \rightarrow Context$ $\qquad$ $\_ + \_ : AExp \times Context \rightarrow Context$

$\_ / \_ : Context \times AExp \rightarrow Context$ $\qquad$ $\_ / \_ : AExp \times Context \rightarrow Context$

$\_ <= \_ : Context \times AExp \rightarrow Context$ $\qquad$ $\_ <= \_ : Int \times Context \rightarrow Context$

$! \_ : Context \rightarrow Context$

$\_ \&\& \_ : Context \times BExp \rightarrow Context$

$\_ = \_ ; : Id \times Context \rightarrow Context$

$\_ \_ : Context \times Stmt \rightarrow Context$

$\texttt{if (\_) \_ else \_} : Context \times Stmt \times Stmt \rightarrow Context$

rules and equations:

$split(Syn) \rightarrow \square[Syn]$ $\qquad$ $plug(\square[Syn]) = Syn$

$split(\langle S, \sigma \rangle) \rightarrow \langle C, \sigma \rangle[Syn]$ **if** $split(S) \rightarrow C[Syn]$

$plug(\langle C, \sigma \rangle[Syn]) = \langle plug(C[Syn]), \sigma \rangle$

$split(A_1 + A_2) \rightarrow (C + A_2)[Syn]$ **if** $split(A_1) \rightarrow C[Syn]$

$plug((C + A_2)[Syn]) = plug(C[Syn]) + A_2$

$split(A_1 + A_2) \rightarrow (A_1 + C)[Syn]$ **if** $split(A_2) \rightarrow C[Syn]$

$plug((A_1 + C)[Syn]) = A_1 + plug(C[Syn])$

$split(A_1 \ / \ A_2) \rightarrow (C \ / \ A_2)[Syn]$ **if** $split(A_1) \rightarrow C[Syn]$

$plug((C \ / \ A_2)[Syn]) = plug(C[Syn]) \ / \ A_2$

$split(A_1 \ / \ A_2) \rightarrow (A_1 \ / \ C)[Syn]$ **if** $split(A_2) \rightarrow C[Syn]$

$plug((A_1 \ / \ C)[Syn]) = A_1 \ / \ plug(C[Syn])$

$split(A_1 <= A_2) \rightarrow (C <= A_2)[Syn]$ **if** $split(A_1) \rightarrow C[Syn]$

$plug((C <= A_2)[Syn]) = plug(C[Syn]) <= A_2$

$split(I_1 <= A_2) \rightarrow (I_1 <= C)[Syn]$ **if** $split(A_2) \rightarrow C[Syn]$

$plug((I_1 <= C)[Syn]) = I_1 <= plug(C[Syn])$

$split(!\ B) \rightarrow (!\ C)[Syn]$ **if** $split(B) \rightarrow C[Syn]$

$plug((!\ C)[Syn]) = !\ plug(C[Syn])$

$split(B_1 \ \&\& \ B2) \rightarrow (C \ \&\& \ B_2)[Syn]$ **if** $split(B_1) \rightarrow C[Syn]$

$plug((C \ \&\& \ B_2)[Syn]) = plug(C[Syn]) \ \&\& \ B_2$

$split(X = A\,;) \rightarrow (X = C\,;)[Syn]$ **if** $split(A) \rightarrow C[Syn]$

$plug((X = C\,;)[Syn]) = X = plug(C[Syn])\,;$

$split(S_1 \ S_2) \rightarrow (C \ S_2)[Syn]$ **if** $split(S_1) \rightarrow C[Syn]$

$plug((C \ S_2)[Syn]) = plug(C[Syn]) \ S_2$

$split(\texttt{if (}B\texttt{)}\ S_1\ \texttt{else}\ S_2) \rightarrow (\texttt{if (}C\texttt{)}\ S_1\ \texttt{else}\ S_2)[Syn]$ **if** $split(B) \rightarrow C[Syn]$

$plug((\texttt{if (}C\texttt{)}\ S_1\ \texttt{else}\ S_2)[Syn]) = \texttt{if (}plug(C[Syn])\texttt{)}\ S_1\ \texttt{else}\ S_2$

Figure 3.37: $\mathcal{R}^{\square}_{\text{RSEC(IMP)}}$: Rewrite logic embedding of IMP evaluation contexts. The implicit split/plug reduction semantics mechanism is replaced by explicit rewrite logic sentences.

$\circ\ Cfg \rightarrow plug(\langle C, \sigma \rangle[\sigma(X)])$ **if** $split(Cfg) \rightarrow \langle C, \sigma \rangle[X] \ \wedge \ \sigma(X) \neq \bot$
$\circ\ Cfg \rightarrow plug(C[I_1 +_{Int} I_2])$ **if** $split(Cfg) \rightarrow C[I_1 + I_2]$
$\circ\ Cfg \rightarrow plug(C[I_1 /_{Int} I_2])$ **if** $split(Cfg) \rightarrow C[I_1 \ / \ I_2] \ \wedge \ I_2 \neq 0$
$\circ\ Cfg \rightarrow plug(C[I_1 \leq_{Int} I_2])$ **if** $split(Cfg) \rightarrow C[I_1 <= I_2]$
$\circ\ Cfg \rightarrow plug(C[\texttt{false}])$ **if** $split(Cfg) \rightarrow C[\texttt{! true}]$
$\circ\ Cfg \rightarrow plug(C[\texttt{true}])$ **if** $split(Cfg) \rightarrow C[\texttt{! false}]$
$\circ\ Cfg \rightarrow plug(C[B_2])$ **if** $split(Cfg) \rightarrow C[\texttt{true \&\&}\ B2]$
$\circ\ Cfg \rightarrow plug(C[\texttt{false}])$ **if** $split(Cfg) \rightarrow C[\texttt{false \&\&}\ B2]$
$\circ\ Cfg \rightarrow plug(\langle C, \sigma[I/X] \rangle[\{\}])$ **if** $split(Cfg) \rightarrow \langle C, \sigma \rangle[X = I\ \texttt{;}] \ \wedge \ \sigma(X) \neq \bot$
$\circ\ Cfg \rightarrow plug(C[S_2])$ **if** $split(Cfg) \rightarrow C[\{\}\ S_2]$
$\circ\ Cfg \rightarrow plug(C[S_1])$ **if** $split(Cfg) \rightarrow C[\texttt{if (true)}\ S_1\ \texttt{else}\ S_2]$
$\circ\ Cfg \rightarrow plug(C[S_2])$ **if** $split(Cfg) \rightarrow C[\texttt{if (false)}\ S_1\ \texttt{else}\ S_2]$
$\circ\ Cfg \rightarrow plug(C[\texttt{if (}B\texttt{) \{}\ S\ \texttt{while (}B\texttt{)}\ S\ \texttt{\} else \{\}}])$ **if** $split(Cfg) \rightarrow C[\texttt{while (}B\texttt{)}\ S\,]$
$\circ \langle \texttt{int}\ Xl\texttt{;}\ S \rangle \rightarrow \langle S, (Xl \mapsto 0) \rangle$

Figure 3.38: $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ — rewrite logic theory corresponding to the first embedding of the reduction semantics with evaluation contexts of IMP.

corresponding to the small-step SOS of IMP in Figure 3.18. The rules that make use of evaluation contexts perform explicit splitting (in the left-hand-side of the condition) and plugging (in the right-hand-side of the conclusion) operations. As already discussed but worth reemphasizing, the main drawbacks of this type of rewrite logic embedding are: (1) the expensive, non-deterministic search involving splitting of the original term is performed for any rule, and (2) it does not take advantage of one of the major optimizations of rewrite engines, indexing, which allows for quick detection of matching rules based on the structure of their left-hand-side terms.

Figure 3.39 shows the rewrite logic theory $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ that follows our second embedding transformation depicted in Figure 3.34. These rules are also added to those corresponding to evaluation contexts in Figure 3.37. Note that now there is precisely one *unconditional* rewrite rule corresponding to each reduction semantics rule of IMP in Figure 3.31 and that, unlike in the first embedding in Figure 3.38, the left-hand-side of each rule preserves the exact structure of the left-hand-side of the original reduction rule (after desugaring of the characteristic rule), so this embedding takes advantage of indexing optimizations in rewrite engines. Like in the first embedding, if a reduction rule does not make use of evaluation contexts, then its corresponding rewrite rule is identical to the rewrite rule corresponding to the small-step SOS embedding discussed in Section 3.3.3 (e.g., the last rule). Unlike in the first embedding, we also need to add a generic conditional rule, the first one in Figure 3.39, which initiates the splitting. We need only one rule of this type because all the left-hand-side terms of reduction rules of IMP in Figure 3.31 that contain a subterm in contextual representation contain that term at the top. As already discussed, if one preferred to write, e.g., the lookup RSEC rule as $\langle c[x], \sigma \rangle \rightarrow \langle c[\sigma(x)], \sigma \rangle$ if $\sigma(x) \neq \bot$, then one would need an additional generic rule, namely $\circ \langle S, \sigma \rangle \rightarrow Cfg'$ **if** $\circ \langle split(S), \sigma \rangle \rightarrow Cfg'$. While these generic rules take care of splitting and can be generated relatively automatically, the remaining rewrite rules that correspond to the reduction rules still make explicit use of the internal (to the embedding) *plug* operation, which can arguably be perceived by language designers as an inconvenience.

Figure 3.40 shows the rewrite logic theory $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ obtained by applying our third embedding (shown in Figure 3.35). These rules are also added to those corresponding to evaluation contexts in Figure 3.37 and, like in the second embedding, there is precisely one unconditional rewrite rule corresponding to each RSEC

$$
\begin{aligned}
\circ\, Cfg &\rightarrow Cfg' \ \textbf{if} \ \circ\, split(Cfg) \rightarrow Cfg' \\
\\
\circ\, \langle C, \sigma\rangle[X] &\rightarrow plug(\langle C, \sigma\rangle[\sigma(X)]) \ \textbf{if} \ \sigma(X) \neq \bot \\
\circ\, C[I_1 + I_2] &\rightarrow plug(C[I_1 +_{Int} I_2]) \\
\circ\, C[I_1 \ / \ I_2] &\rightarrow plug(C[I_1 /_{Int} I_2]) \ \textbf{if} \ I_2 \neq 0 \\
\circ\, C[I_1 \ \texttt{<=} \ I_2] &\rightarrow plug(C[I_1 \leq_{Int} I_2]) \\
\circ\, C[\texttt{!}\ \texttt{true}] &\rightarrow plug(C[\texttt{false}]) \\
\circ\, C[\texttt{!}\ \texttt{false}] &\rightarrow plug(C[\texttt{true}]) \\
\circ\, C[\texttt{true}\ \texttt{\&\&}\ B2] &\rightarrow plug(C[B_2]) \\
\circ\, C[\texttt{false}\ \texttt{\&\&}\ B2] &\rightarrow plug(C[\texttt{false}]) \\
\circ\, \langle C, \sigma\rangle[X = I\,\texttt{;}] &\rightarrow plug(\langle C, \sigma[I/X]\rangle[\{\}]) \ \textbf{if} \ \sigma(X) \neq \bot \\
\circ\, C[\{\}\ S_2] &\rightarrow plug(C[S_2]) \\
\circ\, C[\texttt{if (true)}\ S_1\ \texttt{else}\ S_2] &\rightarrow plug(C[S_1]) \\
\circ\, C[\texttt{if (false)}\ S_1\ \texttt{else}\ S_2] &\rightarrow plug(C[S_2]) \\
\circ\, C[\texttt{while (}B\texttt{)}\ S] &\rightarrow plug(C[\texttt{if (}B\texttt{)}\ \{\, S\ \texttt{while (}B\texttt{)}\ S\,\}\ \texttt{else}\ \{\}]) \\
\circ\, \langle \texttt{int}\ Xl\texttt{;}\ S\rangle &\rightarrow \langle S, (Xl \mapsto 0)\rangle
\end{aligned}
$$

Figure 3.39: $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ — rewrite logic theory corresponding to the second embedding of the reduction semantics with evaluation contexts of IMP.

$$
\begin{aligned}
\circ\, Cfg &\rightarrow Cfg' \ \textbf{if} \ \circ\, plug(split(Cfg)) \rightarrow Cfg' \\
\circ\, C[Syn] &\rightarrow C[Syn'] \ \textbf{if} \ C \neq \Box \wedge \circ\, Syn \rightarrow Syn' \\
\\
\circ\, \langle C, \sigma\rangle[X] &\rightarrow \langle C, \sigma\rangle[\sigma(X)] \ \textbf{if} \ \sigma(X) \neq \bot \\
\circ\, I_1 + I_2 &\rightarrow I_1 +_{Int} I_2 \\
\circ\, I_1 \ / \ I_2 &\rightarrow I_1 /_{Int} I_2] \ \textbf{if} \ I_2 \neq 0 \\
\circ\, I_1 \ \texttt{<=} \ I_2 &\rightarrow I_1 \leq_{Int} I_2 \\
\circ\, \texttt{!}\ \texttt{true} &\rightarrow \texttt{false} \\
\circ\, \texttt{!}\ \texttt{false} &\rightarrow \texttt{true} \\
\circ\, \texttt{true}\ \texttt{\&\&}\ B2 &\rightarrow B_2 \\
\circ\, \texttt{false}\ \texttt{\&\&}\ B2 &\rightarrow \texttt{false} \\
\circ\, \langle C, \sigma\rangle[X = I\,\texttt{;}] &\rightarrow \langle C, \sigma[I/X]\rangle[\{\}] \ \textbf{if} \ \sigma(X) \neq \bot \\
\circ\, \{\}\ S_2 &\rightarrow S_2 \\
\circ\, \texttt{if (true)}\ S_1\ \texttt{else}\ S_2 &\rightarrow S_1 \\
\circ\, \texttt{if (false)}\ S_1\ \texttt{else}\ S_2 &\rightarrow S_2 \\
\circ\, \texttt{while (}B\texttt{)}\ S &\rightarrow \texttt{if (}B\texttt{)}\ \{\, S\ \texttt{while (}B\texttt{)}\ S\,\}\ \texttt{else}\ \{\} \\
\circ\, \langle \texttt{int}\ Xl\texttt{;}\ S\rangle &\rightarrow \langle S, (Xl \mapsto 0)\rangle
\end{aligned}
$$

Figure 3.40: $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ — rewrite logic theory corresponding to the third embedding of the reduction semantics with evaluation contexts of IMP.

rule of IMP. We also need to add a generic conditional rule, the first one, which completely encapsulates the rewrite logic representation of the splitting/plugging mechanism, so that the language designer can next focus exclusively on the semantic rules rather than on their representation in rewrite logic. The second rewrite rule in Figure 3.40 corresponds to the characteristic rule of reduction semantics with evaluation contexts and, as discussed, it is optional; if one includes it, as we did, we think that its definition in Figure 3.40 is as simple and natural as it can be. In what regards the remaining rewrite rules, the only perceivable difference between them and their corresponding reduction rules is that they are preceded by $\circ$.

All the above suggest that, in spite of its apparently advanced context-sensitivity and splitting/plugging mechanism, reduction semantics with evaluation contexts can be safely regarded as a methodological fragment of rewrite logic. Or, put differently, while context-sensitive reduction seems crucial for programming language semantics, it is in fact unnecessary. A conditional rewrite framework can methodologically achieve the same results, and as discussed in this chapter, so can do for the other conventional language semantics approaches.

The following corollary of Theorems 18, 19, and 20 establishes the faithfulness of the representations of the reduction semantics with evaluation contexts of IMP in rewrite logic:

Corollary 7. *For any* IMP *configurations C and C', the following equivalences hold:*

$$\begin{aligned} \text{RSEC(IMP)} \vdash C \to C' &\iff \mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}} \vdash \circ\,\overline{C} \to \overline{C'} \\ &\iff \mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}} \vdash \circ\,\overline{C} \to \overline{C'} \\ &\iff \mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}} \vdash \circ\,\overline{C} \to \overline{C'} \end{aligned}$$

and

$$\begin{aligned} \text{RSEC(IMP)} \vdash C \to^{\star} C' &\iff \mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}} \vdash \star\,\overline{C} \to \star\,\overline{C'} \\ &\iff \mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}} \vdash \star\,\overline{C} \to \star\,\overline{C'} \\ &\iff \mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}} \vdash \star\,\overline{C} \to \star\,\overline{C'} \end{aligned}$$

Therefore, there is no perceivable computational difference between the reduction semantics with evaluation contexts RSEC(IMP) and its corresponding rewrite logic theories.

★ Reduction Semantics with Evaluation Contexts of IMP in Maude

Figure 3.41 shows a Maude representation of the rewrite theory $\mathcal{R}^{\square}_{\text{RSEC(IMP)}}$ in Figure 3.37 that embeds IMP's evaluation contexts by making explicit the split/plug mechanism which is implicit in RSEC. Figure 3.41 also includes the Maude definition of configurations (see Figure 3.36).

We took the freedom to implement a simple optimization which works well in Maude, but which may not work as well in other engines or systems (which is why we did not incorporate it as part of the general procedure to represent reduction semantics with evaluation contexts in rewrite logic): we defined the contextual representation operation `_[_]` to have as result the *kind* (see Section 2.5) `[Syntax]` instead of the sort `Syntax`. This allows us to include the equation `plug(Syn) = Syn`, where `Syn` is a variable of *sort* `Syntax`, which gives us the possibility to also use terms which do not make use of contexts in the right-hand-sides of rewrite rules. To test the rules for splitting, one can write Maude commands such as the one below, asking Maude to search for all splits of a given term:

```
search split(3 <= (2 + X) / 7) =>! Syn:[Syntax] .
```

The `!` tag on the arrow `=>` in the command above tells Maude to only report the normal forms, in this case the completed splits. As expected, Maude finds all seven splits and outputs the following:

```
mod IMP-CONFIGURATIONS-EVALUATION-CONTEXTS is including IMP-SYNTAX + STATE .
  sorts Configuration ExtendedConfiguration .
  subsort Configuration < ExtendedConfiguration .
  op <_,_> : Stmt State -> Configuration .
  op <_> : Pgm -> Configuration .
  ops (o_) (*_) : Configuration -> ExtendedConfiguration [prec 80] .   --- one step
  var Cfg Cfg' : Configuration .
 crl * Cfg => * Cfg' if o Cfg => Cfg' .
endm

mod IMP-SPLIT-PLUG-EVALUATION-CONTEXTS is including IMP-CONFIGURATIONS-EVALUATION-CONTEXTS .
  sorts Syntax Context .  subsorts AExp BExp Stmt Configuration < Syntax .
  op [] : -> Context .  op _[_] : Context Syntax -> [Syntax] [prec 1] .
  ops split plug : Syntax -> Syntax . --- to split Syntax into context[redex]

  var X : Id .  var A A1 A2 : AExp .  var B B1 B2 : BExp .  var S S1 S2 : Stmt .
  var Sigma : State .  var I1 : Int .  var Syn : Syntax .  var C : Context .

  rl split(Syn) => [][Syn] .  eq plug([][Syn]) = Syn .  eq plug(Syn) = Syn .

  op <_,_> : Context State -> Context .  eq plug(< C, Sigma > [Syn]) = < plug(C[Syn]), Sigma > .
 crl split(< S, Sigma >) => < C, Sigma > [Syn] if split(S) => C[Syn] .

  op _+_ : Context AExp -> Context .  eq plug((C + A2)[Syn]) = plug(C[Syn]) + A2 .
 crl split(A1 + A2) => (C + A2)[Syn] if split(A1) => C[Syn] .
  op _+_ : AExp Context -> Context .  eq plug((A1 + C)[Syn]) = A1 + plug(C[Syn]) .
 crl split(A1 + A2) => (A1 + C)[Syn] if split(A2) => C[Syn] .

  op _/_ : Context AExp -> Context .  eq plug((C / A2)[Syn]) = plug(C[Syn]) / A2 .
 crl split(A1 / A2) => (C / A2)[Syn] if split(A1) => C[Syn] .
  op _/_ : AExp Context -> Context .  eq plug((A1 / C)[Syn]) = A1 / plug(C[Syn]) .
 crl split(A1 / A2) => (A1 / C)[Syn] if split(A2) => C[Syn] .

  op _<=_ : Context AExp -> Context .  eq plug((C <= A2)[Syn]) = plug(C[Syn]) <= A2 .
 crl split(A1 <= A2) => (C <= A2)[Syn] if split(A1) => C[Syn] .
  op _<=_ : Int Context -> Context .  eq plug((I1 <= C)[Syn]) = I1 <= plug(C[Syn]) .
 crl split(I1 <= A2) => (I1 <= C)[Syn] if split(A2) => C[Syn] .

  op !_ : Context -> Context .  eq plug((! C)[Syn]) = ! plug(C[Syn]) .
 crl split(! B) => (! C)[Syn] if split(B) => C[Syn] .

  op _&&_ : Context BExp -> Context .  eq plug((C && B2)[Syn]) = plug(C[Syn]) && B2 .
 crl split(B1 && B2) => (C && B2)[Syn] if split(B1) => C[Syn] .

  op _=_; : Id Context -> Context .  eq plug((X = C ;)[Syn]) = X = plug(C[Syn]) ; .
 crl split(X = A ;) => (X = C ;)[Syn] if split(A) => C[Syn] .

  op __ : Context Stmt -> Context .  eq plug((C S2)[Syn]) = plug(C[Syn]) S2 .
 crl split(S1 S2) => (C S2)[Syn] if split(S1) => C[Syn] .

  op if(_)_else_ : Context Stmt Stmt -> Context .
 crl split(if (B) S1 else S2) => (if (C) S1 else S2)[Syn] if split(B) => C[Syn] .
  eq plug((if (C) S1 else S2)[Syn]) = if (plug(C[Syn])) S1 else S2 .
endm
```

Figure 3.41: The configuration and evaluation contexts of IMP in Maude, as needed for the three variants of reduction semantics with evaluation contexts of IMP in Maude.

```
Solution 1 (state 1)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> [][3 <= (2 + X) / 7]

Solution 2 (state 2)
states: 8  rewrites: 19 in ... cpu (. real) (0 rewrites/second)
Syn --> ([] <= (2 + X) / 7)[3]

Solution 3 (state 3)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= [])[(2 + X) / 7]

Solution 4 (state 4)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= ([] / 7))[2 + X]

Solution 5 (state 5)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= (([] + X) / 7))[2]

Solution 6 (state 6)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= ((2 + []) / 7))[X]

Solution 7 (state 7)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= 2 + X / [])[7]
```

If, however, we replace any of the rules for splitting with equations, then, as expected, one looses some of the splitting behaviors. For example, if we replace the generic rule for splitting `rl split(Syn) => [][Syn]` by an apparently equivalent equation `eq split(Syn) = [][Syn]`, then Maude will be able to detect no other splitting of a term t except for $\Box[t]$ (because Maude executes the equations before the rules; see Section 2.5.6).

Figure 3.42 shows two Maude modules implementing the first two rewrite logic theories $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ (Figure 3.38) and $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ (Figure 3.39), and Figure 3.43 shows the Maude module implementing the third rewrite theory $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ (Figure 3.40), respectively. Each of these three Maude modules imports the module `IMP-CONFIGURATION-EVALUATION-CONTEXTS` defined in Figure 3.41 and is executable. Maude, through its rewriting capabilities, therefore yields an IMP reduction semantics with evaluation contexts interpreter for each of the three modules in Figures 3.42 and 3.43. For any of them, the Maude rewrite command

```
rewrite * < sumPgm > .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are also irrelevant, so they were replaced by "..."):

```
rewrites: 42056 in ... cpu (... real) (... rewrites/second)
result ExtendedConfiguration: * < skip, n |-> 0 & s |-> 5050 >
```

The reason for this is ...

One can use any of the general-purpose tools provided by Maude on the reduction semantics with evaluation contexts definitions above. For example, one can exhaustively search for all possible behaviors of a program using the `search` command:

```
mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS .
  var X : Id .  var I I1 I2 : Int .  var B B2 : BExp .  var S S1 S2 : Stmt .
  var Xl : List{Id} .  var Sigma : State .  var Cfg : Configuration .  var C : Context .

 crl o Cfg => plug(< C,Sigma >[Sigma(X)]) if split(Cfg) => < C,Sigma >[X]
  /\ Sigma(X) =/=Bool undefined .
 crl o Cfg => plug(C[I1 +Int I2]) if split(Cfg) => C[I1 + I2] .
 crl o Cfg => plug(C[I1 /Int I2]) if split(Cfg) => C[I1 / I2]
  /\ I2 =/=Bool 0 .
 crl o Cfg => plug(C[I1 <=Int I2]) if split(Cfg) => C[I1 <= I2] .
 crl o Cfg => plug(C[false]) if split(Cfg) => C[! true] .
 crl o Cfg => plug(C[true]) if split(Cfg) => C[! false] .
 crl o Cfg => plug(C[B2]) if split(Cfg) => C[true && B2] .
 crl o Cfg => plug(C[false]) if split(Cfg) => C[false && B2] .
 crl o Cfg => plug(C[S]) if split(Cfg) => C[{S}] .
 crl o Cfg => plug(< C,Sigma[I / X] >[{}]) if split(Cfg) => < C,Sigma >[X = I ;]
  /\ Sigma(X) =/=Bool undefined .
 crl o Cfg => plug(C[S2]) if split(Cfg) => C[{} S2] .
 crl o Cfg => plug(C[S1]) if split(Cfg) => C[if (true) S1 else S2] .
 crl o Cfg => plug(C[S2]) if split(Cfg) => C[if (false) S1 else S2] .
 crl o Cfg => plug(C[if (B) {S while (B) S} else {}]) if split(Cfg) => C[while (B) S] .
  rl o < int Xl ; S > => < S,(Xl |-> 0) > .
endm



mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS .
  var X : Id .  var I I1 I2 : Int .  var B B2 : BExp .  var S S1 S2 : Stmt .
  var Xl : List{Id} .  var Sigma : State .  var Cfg Cfg' : Configuration .  var C : Context .

 crl o Cfg => Cfg' if o split(Cfg) => Cfg' .   --- generic rule enabling splitting

 crl o < C,Sigma >[X] => plug(< C,Sigma >[Sigma(X)])
  if Sigma(X) =/=Bool undefined .
  rl o C[I1 + I2] => plug(C[I1 +Int I2]) .
 crl o C[I1 / I2] => plug(C[I1 /Int I2])
  if I2 =/=Bool 0 .
  rl o C[I1 <= I2] => plug(C[I1 <=Int I2]) .
  rl o C[! true] => plug(C[false]) .
  rl o C[! false] => plug(C[ true]) .
  rl o C[true && B2] => plug(C[B2]) .
  rl o C[false && B2] => plug(C[false]) .
  rl o C[{S}] => plug(C[S]) .
 crl o < C,Sigma >[X = I ;] => plug(< C,Sigma[I / X] >[{}])
  if Sigma(X) =/=Bool undefined .
  rl o C[{} S2] => plug(C[S2]) .
  rl o C[if (true) S1 else S2] => plug(C[S1]) .
  rl o C[if (false) S1 else S2] => plug(C[S2]) .
  rl o C[while (B) S] => plug(C[if (B) {S while (B) S} else {}]) .
  rl o < int Xl ; S > => < S,(Xl |-> 0) > .
endm
```

Figure 3.42: The first two reduction semantics with evaluation contexts of IMP in Maude.

```
mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS .
  var X : Id .  var I I1 I2 : Int .  var B B2 : BExp .  var S S1 S2 : Stmt .  var Xl : List{Id} .
  var Sigma : State .  var Cfg Cfg' : Configuration .  var Syn Syn' : Syntax  .  var C : Context .

  crl o Cfg => Cfg' if plug(o split(Cfg)) => Cfg' .           --- generic rule enabling splitting
  crl o C[Syn] => C[Syn'] if C =/=Bool [] /\ o Syn => Syn' .  --- characteristic rule

 crl o < C,Sigma >[X] => < C,Sigma >[Sigma(X)]
  if Sigma(X) =/=Bool undefined .
  rl o I1 + I2 => I1 +Int I2 .
 crl o I1 / I2 => I1 /Int I2
  if I2 =/=Bool 0 .
  rl o I1 <= I2 => I1 <=Int I2 .
  rl o ! true => false .
  rl o ! false => true  .
  rl o true && B2 => B2 .
  rl o false && B2 => false .
  rl o {S} => S .
 crl o < C,Sigma >[X = I ;] => < C,Sigma[I / X] >[{}]
  if Sigma(X) =/=Bool undefined .
  rl o {} S2 => S2 .
  rl o if (true) S1 else S2 => S1 .
  rl o if (false) S1 else S2 => S2 .
  rl o while (B) S => if (B) {S while (B) S} else {} .
  rl o < int Xl ; S > => < S,(Xl |-> 0) > .
endm
```

Figure 3.43: The third reduction semantics with evaluation contexts of IMP in Maude.

```
search * < sumPgm > =>! Cfg:ExtendedConfiguration .
```

As expected, only one behavior will be discovered because our IMP language so far is deterministic. Not unexpectedly, the same number of states as in the case of small-step SOS and MSOS will be discovered by this search command, namely 1709. Indeed, the splitting/cooling mechanism of RSEC is just another way to find where the next reduction step should take place; it does not generate any different reductions of the original configuration.

3.7.4 Notes

Reduction semantics with evaluation contexts was introduced by Felleisen and his collaborators (see, e.g., [25, 88]) as a variant small-step structural operational semantics. By making the evaluation context explicit and modifiable, reduction semantics with evaluation contexts is considered by many to be a significant improvement over small-step SOS. Like small-step SOS, reduction semantics with evaluation contexts has been broadly used to give semantics to programming languages and to various calculi. We here only briefly mention some strictly related work.

How expensive is the splitting of a term into an evaluation context and a redex? Unfortunately, it cannot be more efficient than testing the membership of a word to a context-free grammar and the latter is expected to be cubic in the size of the original term (folklore). Indeed, consider G an arbitrary CFG whose start symbol is S and let G_C be the "evaluation context" CFG grammar adding a fresh "context" nonterminal C, a fresh terminal #, and productions $C \rightarrow \Box \mid CS\#$. Then it is easy to see that a word α is in the language of G if and only if $\#\alpha\#$ can be split as a contextual representation (can only be $(\Box\alpha\#)[\#]$). Thus, we should expect, in the worst case, a *cubic* complexity to split a term into an evaluation context and a redex. An additional exponent needs to be added, thus making splitting expected to be a *quadratic* operation in the worst case, when nested contexts are allowed in rules (i.e., when the redex is itself a contextual representation). Unfortunately, this terrible complexity needs to be paid at each step of reduction, not to mention that the size of the program to reduce can also grow as it is reduced. One possibility to decrease this complexity is to attempt to incrementally compute at each step the evaluation context that is needed at the next step (like in refocusing; see below); however, in the worst case the right-hand-sides of rules may contain no contexts, in which case a fresh split is necessary at each step.

Besides our own efforts, we are aware of three other attempts to develop executable engines for reduction semantics with evaluation contexts, which we discuss here in chronological order:

1. A specification language for syntactic theories with evaluation contexts is proposed by Xiao *et al.* [90, 89], together with a system which generates Ocaml interpreters from specifications. Although the compiler in [90, 89] is carefully engineered, as rightfully noticed by Danvy and Nielsen in [20] it cannot avoid the quadratic overhead due to the context-decomposition step. This is consistent with our own observations expressed at several places in this section, namely that the advanced parsing underlying reduction semantics with evaluation contexts is the most expensive part when one is concerned with execution. Fortunately, the splitting of syntax into context and redex can be and typically is taken for granted in theoretical developments, making abstraction of the complexity of its implementation.

2. A technique called *refocusing* is proposed by Danvy and Nielsen in [20, 19]. The idea underlying refocusing is to keep the program decomposed at all times (in a first-order continuation-like form) and to perform minimal changes to the resulting structure to find the next redex. Unfortunately, refocusing appears to work well only with restricted RSEC definitions, namely ones whose evaluation contexts grammar has the property of unique decomposition of a term into a context and a redex (so constructs like the non-deterministic addition of IMP are disallowed), and whose reduction rules are deterministic.

3. PLT-Redex, which is implemented in Scheme by Findler and his collaborators [38, 24], is perhaps the most advanced tool developed specifically to execute reduction semantics with evaluation contexts. PLT-Redex builds upon a direct implementation of context-sensitive reduction, so it cannot avoid the worst-case quadratic complexity of context decomposition, same as the interpreters generated by the system in [90, 89] discussed above. Several large language semantics engineering case studies using PLT-Redex are discussed in [24].

Our embeddings of reduction semantics with evaluation contexts into rewrite logic are inspired from a related embedding by Șerbănuță *et al.* in [74]. The embedding in [74] was similar to our third embedding here, but it included splitting rules also for terms in reducible form, e.g., $split(I_1 <= I_2) \rightarrow \square[I_1 <= I_2]$. Instead, we preferred to include a generic rule $split(Syn) \rightarrow \square[Syn]$ here, which allows us to more mechanically derive the rewrite rules for splitting from the CFG of evaluation contexts. Calculating the exact complexity of our approach seems to be hard, mainly because of optimizations employed by rewrite engines, e.g., indexing. Since at each step we still search for all the relevant splits of the term into an evaluation context and a redex, in the worst case we still pay the quadratic complexity. However, as suggested by the performance numbers in [74] comparing Maude running the resulting rewrite theory against PLT-Redex, which favor the former by a large margin, our embeddings may serve as alternative means to getting more efficient implementations of reduction semantics engines. There are strong reasons to believe that our third embedding can easily be automated in a way that the user never sees the split/plug operations.

3.7.5 Exercises

Exercise 127. *Suppose that one does not like mixing semantic components with syntactic evaluation contexts as we did above (by including the production Context ::= ⟨Context, State⟩). Instead, suppose that one prefers to work with configuration tuples like in SOS, holding the various components needed for the language semantics, the program or fragment of program being just one of them. In other words, suppose that one wants to make use of the contextual representation notation only on the syntactic component of configurations. In this case, the characteristic rule becomes*

$$\frac{\langle e, \gamma \rangle \rightarrow \langle e', \gamma' \rangle}{\langle c[e], \gamma \rangle \rightarrow \langle c[e'], \gamma' \rangle}$$

where γ and γ' consist of configuration semantic components that are necessary to evaluate e and e′, respectively, such as states, outputs, stacks, etc. Modify accordingly the six reduction semantics with evaluation contexts rules discussed at the beginning of Section 3.7.

The advantage of this approach is that it allows the evaluation contexts to be defined exclusively over the syntax of the language. However, configurations holding code and state still need to be defined. Moreover, many rules which looked compact before, such as $i_1 <= i_2 \rightarrow i_1 \leq_{Int} i_2$, will now look heavier, e.g., $\langle i_1 <= i_2, \sigma \rangle \rightarrow \langle i_1 \leq_{Int} i_2, \sigma \rangle$.

Exercise 128. *Like in Exercise 127, suppose that one does not like to mix syntactic and semantic components in evaluation contexts, but that, instead, one is willing to accept to slightly enrich the syntax of the programming language with a special statement construct "Stmt ::= `int` Id = AExp `;`" which both declares and initializes a variable*[10]*. Then*

1. *Write structural identities that desugar the current top-level program variable declarations `int` $x_1, ..., x_n$ `;` s into statements of the form `int` x_1 = 0 `;` ... `int` x_n = 0 `;` s.*

[10] Similar language constructs exist in many programming language (C, Java, etc.).

2. *Add a new context production that allows evaluation after the new variable declarations.*

3. *Modify the variable lookup and assignment rules discussed above so that one uses the new declarations instead of a state. Hint: the context should have the form "*`int` x `=` i`;` c*".*

The advantage of this approach is that one does not need an explicit state anymore, so the resulting definition is purely syntactic. In fact, the state is there anyway, but encoded syntactically as a sequence of variable initializations preceding any other statement. This trick works in this case, but it cannot be used as a general principle to eliminate configurations in complex languages.

Exercise* 129. *Exercise 127 suggests that one can combine MSOS (Section 3.6) and evaluation contexts, in that one can use MSOS's labels to obtain modularity at the configuration level and one can use the evaluation contexts idea to detect and modify the contexts/redexes in the syntactic component of a configuration. Rewrite the six rules discussed at the beginning of Section 3.7 as they would appear in a hypothetical framework merging MSOS and evaluation contexts.*

Exercise 130. *Modify the reduction semantics with evaluation contexts of* IMP *in Figures 3.30 and 3.31 so that / short-circuits when its numerator evaluates to* 0*.*
Hint: Make / strict in only the first argument, then use a rule to reduce $0 \,/\, a_2$ *to 0 and a rule to reduce* $i_1 \,/\, a_2$ *to* $i_1 \,/\text{'}\, a_2$ *when* $i_1 \neq 0$*, where /' is strict in its second argument, and finally a rule to reduce* $i_1 \,/\text{'}\, i_2$ *to* $i_1 \,/_{Int}\, i_2$ *when* $i_2 \neq 0$*.*

Exercise 131. *Modify the reduction semantics with evaluation contexts of* IMP *in Figures 3.30 and 3.31 so that conjunction is not short-circuited anymore but, instead, is non-deterministically strict in both its arguments.*

Exercise 132. *Give an alternative reduction semantics of* IMP *with evaluation contexts following the approach in Exercise 127 (that is, use evaluation contexts only for the* IMP *language syntax, and handle the semantic components using configurations, like in SOS).*

Exercise 133. *Give an alternative reduction semantics of* IMP *with evaluation contexts following the approach in Exercise 128.*

Exercise* 134. *Give a semantics of* IMP *using the hypothetical framework combining reduction semantics with evaluation contexts and MSOS proposed in Exercise 129.*

Exercise 135. *Modify the rewrite theory* $\mathcal{R}^{\Box}_{\text{RSEC(IMP)}}$ *in Figure 3.37 so that later on one can define the reduction semantics of / to short-circuit when the numerator evaluates to* 0 *(as required in Exercises 137, 143, and 149).*

Exercise 136. *Modify the rewrite theory* $\mathcal{R}^{\Box}_{\text{RSEC(IMP)}}$ *in Figure 3.37 so that one can later on define the reduction semantics of conjunction to be non-deterministically strict in both its arguments (as required in Exercises 138, 144, and 150).*

Exercise 137. *Modify the rewrite theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the reduction semantics of / that short-circuits when the numerator evaluates to* 0 *(see also Exercise 135).*

Exercise 138. *Modify the rewrite theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the reduction semantics of conjunction that defines it as non-deterministically strict in both its arguments (see also Exercise 136).*

Exercise 139. *As discussed in several places so far in Section 3.7, the reduction semantics rules for variable lookup and assignment can also be given in a way in which their left-hand-side terms are not in contextual representation (i.e.,* $\langle c[x], \sigma \rangle$ *instead of* $\langle c, \sigma \rangle[x]$*, etc.). Modify the corresponding rewrite rules of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for this alternative reduction semantics.*

Exercise 140. *Modify the rewrite logic theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the alternative reduction semantics with evaluation contexts of* IMP *in Exercise 132.*

Exercise 141. *Modify the rewrite logic theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the alternative reduction semantics with evaluation contexts of* IMP *in Exercise 133.*

Exercise* 142. *Combining the underlying ideas of the embedding of MSOS in rewrite logic discussed in Section 3.6.3 and the embedding of reduction semantics with evaluation contexts in Figure 3.33, give a rewrite logic semantics of* IMP *corresponding to the semantics of* IMP *in Exercise 134.*

Exercise 143. *Same as Exercise 137, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 144. *Same as Exercise 138, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 145. *Same as Exercise 139, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 146. *Same as Exercise 140, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 147. *Same as Exercise 141, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise* 148. *Same as Exercise 142, but for Figure 3.39 (instead of Figure 3.38).*

Exercise 149. *Same as Exercise 137, but for* $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 150. *Same as Exercise 138, but for* $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 151. *Same as Exercise 139, but for* $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 152. *Same as Exercise 140, but for* $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise 153. *Same as Exercise 141, but for* $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

Exercise* 154. *Same as Exercise 142, but for Figure 3.40 (instead of Figure 3.38).*

Exercise 155. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 so that / short-circuits when its numerator evaluates to* 0 *(see also Exercises 130, 135, 137, 143, and 149).*

Exercise 156. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 so that conjunction is not short-circuited anymore but, instead, is non-deterministically strict in both its arguments (see also Exercises 131, 136, 138, 144, and 150).*

Exercise 157. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the alternative reduction semantics in Exercises 139, 145, and 151.*

Exercise 158. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the alternative reduction semantics in Exercises 140, 146, and 152.*

Exercise 159. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the alternative reduction semantics in Exercises 141, 147, and 153.*

Exercise⋆ 160. *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the semantics in Exercises 142, 148, and 154.*

Exercise 161. *Same as Exercise 86, but for reduction semantics with evaluation contexts instead of small-step SOS: add variable increment to* IMP*, like in Section 3.7.2.*

Exercise 162. *Same as Exercise 90, but for reduction semantics with evaluation contexts instead of small-step SOS: add input/output to* IMP*, like in Section 3.7.2.*

Exercise⋆ 163. *Consider the hypothetical framework combining MSOS with reduction semantics with evaluation contexts proposed in Exercise 129, and in particular the* IMP *semantics in such a framework in Exercise 134, its rewrite logic embeddings in Exercises 142, 148, and 154, and their Maude implementation in Exercise 160. Define the semantics of the input/output constructs above modularly first in the framework in discussion, then using the rewrite logic embeddings, and finally in Maude.*

Exercise 164. *Same as Exercise 95, but for reduction semantics with evaluation contexts instead of small-step SOS: add abrupt termination to* IMP*, like in Section 3.7.2.*

Exercise 165. *Same as Exercise 104, but for reduction semantics with evaluation contexts instead of small-step SOS: add dynamic threads to* IMP*, like in Section 3.7.2.*

Exercise 166. *Same as Exercise 109, but for reduction semantics with evaluation contexts instead of small-step SOS: add local variables using* `let` *to* IMP*, like in Section 3.7.2.*

Exercise⋆ 167. *This exercise asks to define* IMP++ *in reduction semantics, in various ways. Specifically, redo Exercises 114, 115, 116, 117, and 118, but for the reduction semantics with evaluation contexts of* IMP++ *discussed in Section 3.7.2 instead of its small-step SOS in Section 3.5.6.*

3.8 The Chemical Abstract Machine (CHAM)

The *chemical abstract machine*, or the *CHAM*, is both a model of concurrency and a specific operational semantics style. The states of a CHAM are metaphorically regarded as chemical solutions formed with floating molecules. Molecules can interact with each other by means of reactions. A reaction can involve several molecules and can change them, delete them, and/or create new molecules. One of the most appealing aspects of the chemical abstract machine is that its reactions can take place concurrently, unrestricted by context. To facilitate local computation and to represent complex data-structures, molecules can be nested by encapsulating groups of molecules as sub-solutions. The chemical abstract machine was proposed as an alternative to SOS and its variants, including reduction semantics with evaluation contexts, in an attempt to circumvent their limitations, particularly their lack of support for true concurrency.

CHAM Syntax

The *basic molecules* of a CHAM are ordinary algebraic terms over a user-defined syntax. Several molecules wrapped within a membrane form a *solution*, which is also a molecule. The CHAM uses the symbols $\{\!|$ and $|\!\}$ to denote membranes. For example, $\{\!|m_1\ m_2\ \ldots\ m_k|\!\}$ is a solution formed with the molecules $m_1, m_2, \ldots, m_k$. The order of molecules in a solution is irrelevant, so a solution can be regarded as a multi-set (or bag) of molecules wrapped within a membrane. Since solutions are themselves molecules, we can have arbitrarily nested molecules. This nesting mechanism is generic for all CHAMs and has the following (algebraic) syntax:

$$\begin{array}{rcl} \mathit{Molecule} & ::= & \mathit{Solution} \mid \mathit{Molecule} \triangleleft \mathit{Solution} \\ \mathit{Solution} & ::= & \{\!|\mathbf{Bag}\{\mathit{Molecule}\}|\!\} \end{array}$$

The operator $\_ \triangleleft \_$ is called the *airlock* operator and will be discussed shortly (under general CHAM laws), after we discuss the CHAM rules. When defining a CHAM, one is only allowed to extend the syntax of molecules, which implicitly also extends the syntax that the solution terms can use. However, one is not allowed to explicitly extend the syntax of solutions. In other words, solutions can only be built using the generic syntax above, on top of user-defined syntactic extensions of molecules. Even though we do not formalize it here (and we are not aware of other formulations elsewhere either), it is understood that one can have multiple types of molecules in a CHAM.

Specific CHAM Rules

In addition to extending the syntax of molecules, a CHAM typically also defines a set of *rules*, each rule being a rule schemata but called a rule for simplicity. A CHAM rule has the form

$$m_1\ m_2\ \ldots\ m_k \rightarrow m'_1\ m'_2\ \ldots m'_l$$

where $m_1, m_2, \ldots, m_k$ and $m'_1, m'_2, \ldots, m'_l$ are not necessarily distinct molecules (since CHAM rules are schemata, these molecule terms may contain meta-variables). Molecules appearing in a rule are restricted to contain only subsolution terms which are either solution meta-variables or otherwise have the form $\{\!|m|\!\}$, where m is some molecule term. For example, a CHAM rule cannot contain subsolution terms of the form $\{\!|m\ s|\!\}$, $\{\!|m_1\ m_2|\!\}$, or $\{\!|m_1\ m_2\ s|\!\}$, with m, m_1, m_2 molecule terms and s solution term, but it can contain ones of the form $\{\!|m|\!\}$, $\{\!|m_1 \triangleleft \{\!|m_2|\!\}|\!\}$, $\{\!|m \triangleleft s|\!\}$, etc. This restriction is justified by chemical intuitions, namely that matching inside a solution is a rather complex operation which needs special handling (the airlock operator $\_ \triangleleft \_$ is used for this purpose). Note that CHAM rules are unconditional, that is, they have no premises.

General CHAM Laws

Any chemical abstract machine obeys the four laws below. Let CHAM be[11] a chemical abstract machine. Below we assume that mol, mol', mol_1, etc., are arbitrary concrete molecules of CHAM (i.e., no meta-variables) and that sol, sol', etc., are concrete solutions of it. If sol is the solution $\{\!|mol_1\ mol_2\ \ldots\ mol_k|\!\}$ and sol' is the solution $\{\!|mol'_1\ mol'_2\ \ldots\ mol'_l|\!\}$, then $sol \uplus sol'$ is the solution $\{\!|mol_1\ mol_2\ \ldots\ mol_k\ mol'_1\ mol'_2\ \ldots\ mol'_l|\!\}$.

1. ***The Reaction Law.*** Given a CHAM rule

$$m_1\ m_2\ \ldots\ m_k \rightarrow m'_1\ m'_2\ \ldots m'_l \quad \in \quad \text{CHAM}$$

if $mol_1, mol_2, \ldots, mol_k$ and $mol'_1, mol'_2, \ldots, mol'_l$ are (concrete) instances of $m_1\ m_2\ \ldots\ m_k$ and of $m'_1\ m'_2\ \ldots m'_l$ by a common substitution, respectively, then

$$\text{CHAM} \vdash \{\!|mol_1\ mol_2\ \ldots\ mol_k|\!\} \rightarrow \{\!|mol'_1\ mol'_2\ \ldots mol'_l|\!\}$$

2. ***The Chemical Law.*** Reactions can be performed freely within any solution:

$$\frac{\text{CHAM} \vdash sol \rightarrow sol'}{\text{CHAM} \vdash sol \uplus sol'' \rightarrow sol' \uplus sol''}$$

3. ***The Membrane Law.*** A subsolution can evolve freely in any solution context $\{\!|cxt[\Box]|\!\}$:

$$\frac{\text{CHAM} \vdash sol \rightarrow sol'}{\text{CHAM} \vdash \{\!|cxt[sol]|\!\} \rightarrow \{\!|cxt[sol']|\!\}}$$

4. ***The Airlock Law.***

$$\text{CHAM} \vdash \{\!|mol|\!\} \uplus sol \leftrightarrow \{\!|mol \triangleleft sol|\!\}$$

Note the unusual fact that $m_1\ m_2\ \ldots\ m_k \rightarrow m'_1\ m'_2\ \ldots m'_l$ being a rule in CHAM does not imply that $\text{CHAM} \vdash m_1\ m_2\ \ldots\ m_k \rightarrow m'_1\ m'_2\ \ldots m'_l$. Indeed, the CHAM rules are regarded as descriptors of changes that can take place in solutions and only in solutions, while CHAM sequents are incarnations of those otherwise purely abstract rules. What may be confusing is that the same applies also when k and l (the numbers of molecules in the left-hand and right-hand-sides of the CHAM rule) happen to be 1 and m_1 and m'_1 happen to be solutions that contain no meta-variables. The two $\rightarrow$ arrows, namely the one in CHAM rules and the one in CHAM sequents, ought to be different symbols; however, we adhere to the conventional CHAM notation which uses the same symbol for both. Moreover, when the CHAM is clear from context, we also follow the conventional notation and drop it from sequents, that is, we write $sol \rightarrow sol'$ instead of $\text{CHAM} \vdash sol \rightarrow sol'$. While we admit that these conventions may sometimes be confusing, in that $sol \rightarrow sol'$ can be a rule or a sequent or even both, we hope that the context makes it clear which one is meant.

The Reaction Law says that CHAM rules can only apply in solutions (wrapped by a membrane), and not arbitrarily wherever they match. The Chemical Law says that once a reaction take place in a certain solution, it can take place in any other larger solution. In other words, the fact that a solution has more molecules than required by the rule does not prohibit the rule from applying. The Reaction and the Chemical laws together say that CHAM rules can apply inside any solutions having some molecules that match the left-had side

[11] To avoid inventing new names, it is common to use CHAM both as an abbreviation for "the chemical abstract machine" and as a name of an arbitrary but fixed chemical abstract machine.

of the CHAM rule. An interesting case is when the left-hand-side term of the CHAM rule has only one molecule, i.e., when $k = 1$, because the CHAM rule is still allowed to only apply within a solution; it cannot apply in other places where the left-hand-side happens to match.

The Membrane Law says that reactions can take place in any solution context. Indeed, $\{\!|cxt[sol]|\!\}$ says that the solution *sol* (which is wrapped in a membrane) appears somewhere, anywhere, inside a solution context $\{\!|cxt[\Box]|\!\}$. Here *cxt* can be any bag-of-molecule context, and we write $cxt[\Box]$ to highlight the fact that it is a context with a hole $\Box$. By wrapping $cxt[\Box]$ in a membrane we enforce a solution context. This rule also suggests that, at any given moment, the global term to rewrite using the CHAM rules should be a solution. Indeed, the CHAM rewriting process gets stuck as soon as the term becomes a proper molecule (not a solution), because the Membrane Law cannot apply.

The Airlock Law is reversible (i.e., it comprises two rewrite rules, one from left-to-right and one from right-to-left) and it allows to extract a molecule from a solution, putting the rest of the solution within a membrane. Using this law one can, for example, rewrite a solution $\{\!|mol_1\ mol_2\ \ldots, mol_k|\!\}$ into $\{\!|mol_1 \triangleleft \{\!|mol_2\ \ldots\ mol_k|\!\}|\!\}$. The advantage of doing so is that one can now match the molecule mol_1 within other rules. Indeed, recall that sub-solutions that appear in rules cannot specify any particular molecule term among the rest of the solution, unless the solution contains precisely that molecule. Since $mol_1 \triangleleft \{\!|mol_2\ \ldots\ mol_k|\!\}$ is a molecule, $\{\!|mol_1 \triangleleft \{\!|mol_2\ \ldots\ mol_k|\!\}|\!\}$ can match molecule terms of the form $\{\!|m \triangleleft s|\!\}$ appearing in CHAM rules, this way one effectively matching (and possibly modifying) the molecule mol_1 via the specific CHAM rules. The Airlock Law is the only means provided by the CHAM to extract or put molecules in a solution.

The four laws above do not completely define the CHAM rewriting; they are only properties that the CHAM rewriting should satisfy. In particular, they do not capture the concurrency potential of the CHAM.

Definition 23. *The four laws above give us a proof system for CHAM sequents. As usual,* $\text{CHAM} \vdash sol \to sol'$ *in isolation means mean that it is derivable. Also, let* $\to^*$ *denote the reflexive and transitive closure of* $\to$*, that is,* $\text{CHAM} \vdash sol \to^* sol'$ *iff* $sol = sol'$ *or there is some* sol'' *such that* $\text{CHAM} \vdash sol \to sol''$ *and* $\text{CHAM} \vdash sol'' \to^* sol'$*. Finally,* $\text{CHAM} \vdash sol \leftrightarrow sol'$ *is a shorthand for the seqents* $\text{CHAM} \vdash sol \to sol'$ *and* $\text{CHAM} \vdash sol' \to sol$*, and we say that it is derivable iff the two sequents are derivable.*

None of the two sequents in Definition 23 captures the underlying concurrent computation of the CHAM. Indeed, $\text{CHAM} \vdash sol \to sol'$ says that one and only one reaction takes place somewhere in *sol*, while $\text{CHAM} \vdash sol \to^* sol'$ says that arbitrarily many steps take place, including ones which can be done concurrently but also ones which can only take place sequentially. Therefore, we can think of the four laws above, and implicitly of the sequents $\text{CHAM} \vdash sol \to sol'$ and $\text{CHAM} \vdash sol \to^* sol'$, as expressing the descriptive capability of the CHAM: what is possible and what is not possible to compute using the CHAM, and not how it operates. Nevertheless, it is recommended to think of CHAM reactions as taking place concurrently whenever they do not involve the same molecules, even though this "concurrent reaction" notion is not formalized here. We are actually not aware of any works that formalize the CHAM concurrency.

A common source of misunderstanding the CHAM is to wrongly think of CHAM rules as ordinary rewriting rules modulo the associativity, commutativity and identity of the molecule grouping (inside a solution) operation. The major distinction between CHAM rules and such rewrite rules is that the former only apply within solutions (i.e., wrapped by membranes) no matter whether the rule contains one or more molecules in its left-hand or right-hand terms, while the latter apply anywhere they match. For example, supposing that we extend the syntax of molecules with the syntax of IMP in Section 3.1.1 and add a CHAM rule $m + 0 \to m$, then we can rewrite the solution $\{\!|(3 + 0)\ 7|\!\}$ to solution $\{\!|3\ 7|\!\}$, but we cannot rewrite the molecule 5 / (3 + 0) to molecule 5 / 3 regardless of what context it is in, because 3 / 0 is not in a solution. We cannot even rewrite the isolated (i.e., not in a solution context) term 3 + 0 to 3 in CHAM, for the same reason.

Classification of CHAM Rules

The rules of a CHAM are typically partitioned into three intuitive categories, namely *heating*, *cooling* and *reaction* rules, although there are no formal requirements imposing a rule to be into one category or another. Moreover, the same laws discussed above apply the same way to all categories of rules, and the same restrictions preventing multiset matching apply to all of them.

- *Heating* rules, distinguished by using the relation symbol $\rightharpoonup$ instead of $\rightarrow$, are used to structurally rearrange the solution so that reactions can take place.
- *Cooling* rules, distinguished by using the relation symbol $\rightharpoondown$ instead of $\rightarrow$, are used after reactions take place to structurally rearrange the solution back into a convenient form, including to remove useless molecules or parts of them.
- *Reaction* rules, which capture the intended computational steps and use the conventional rewrite symbol $\rightarrow$, are used to evolve the solution in an irreversible way.

The heating and cooling rules can typically be paired, with each heating rule $l \rightharpoonup r$ having a symmetric cooling rule $l \leftharpoondown r$, so that we can view them as a single bidirectional *heating/cooling* rule. The CHAM notation for writing such heating/cooling rules is the following:

$$l \rightleftharpoons r$$

In particular, it makes sense to regard the airlock axiom as an example of such a heating/cooling bidirectional rule, that is,

$$\{\!|m_1\ m_2\ \ldots\ m_k|\!\} \rightleftharpoons \{\!|m_1 \triangleleft \{\!|m_2\ \ldots\ m_k|\!\}|\!\}$$

where m_1, m_2, ..., m_k are molecule meta-variables. The intuition here is that we can heat the solution to extract m_1 in an airlock, or we can cool it down so that the airlock m_1 is diffused within the solution. However, we need to assume one such rule for each $k > 0$.

As one may expect, the reaction rules are the heart of the CHAM and properly correspond to state transitions. The heating and cooling rules express *structural rearrangements*, so that the reaction rules can match and apply. In other words, we can view the reaction rules as being applied *modulo* the heating and cooling rules. We are going to suggestively use the notation CHAM $\vdash sol \rightharpoonup sol'$, respectively CHAM $\vdash sol \rightharpoondown sol'$ whenever the rewrite step taking sol to sol' is a heating rule, respectively a cooling rule. Similarly, we may use the notations CHAM $\vdash sol \rightharpoonup^* sol'$ and CHAM $\vdash sol \rightharpoondown^* sol'$ for the corresponding reflexive/transitive closures. Also, to emphasize the fact that there is only one reaction rule applied, we take the freedom to (admittedly ambiguously) write CHAM $\vdash sol \rightarrow sol'$ instead of CHAM $\vdash sol(\rightarrow \cup \rightharpoonup \cup \rightharpoondown)^* sol'$ whenever all the involved rules but one are heating or cooling rules.

3.8.1 The CHAM of IMP

We next show how to give IMP a CHAM semantics. CHAM is particularly well-suited to giving semantics to concurrent distributed calculi and languages, yielding considerably simpler definitions than those afforded by SOS. Since IMP is sequential, it cannot take full advantage of the CHAM's true concurrency capabilities; the multi-threaded IMP++ language discussed in Section 3.5 will make better use of CHAM's capabilities. Nevertheless, some of CHAM's capabilities turn out to be useful even in this sequential language application, others turn out to be deceiving. Our CHAM semantics for IMP below follows in principle the reduction semantics with evaluation contexts definition discussed in Section 3.7.1. One can formally show that a step

performed using reduction under evaluation contexts is equivalent to a suite of heating steps, followed by one reaction step, followed by a suite of cooling steps.

The CHAM defined below is just one possible way to give IMP a CHAM semantics. CHAM, like rewriting, is a general framework which does not impose upon its users any particular definitional style. In our case, we chose to conceptually distinguish two types of molecules; we say "conceptually" because, for simplicity, we prefer to define only one *Molecule* syntactic category in our CHAM:

- *Syntactic molecules*, which include all the syntax of IMP in Section 3.1.1, plus all the syntax of its evaluation contexts in Section 3.7.1, plus a mechanism to flatten evaluation contexts; for simplicity, we prefer to not include a distinct type of molecule for each distinct syntactic category of IMP.
- *State molecules*, which are pairs $x \mapsto i$, where $x \in \mathit{Id}$ and $i \in \mathit{Int}$.

For clarity, we prefer to keep the syntactic and the state molecules in separate solutions. More precisely, we work with top-level configurations which are solutions of the form

$$\{\!\{\mathit{Syntax}\}\!\}\ \ \{\!\{\mathit{State}\}\!\}$$

Syntax and *State* are solutions containing syntactic and state molecules, respectively. For example,

$$\{\!\{x = 3 \,/\, (x + 2)\,;\}\!\}\ \{\!\{x \mapsto 1\ \ y \mapsto 0\}\!\}$$

is a CHAM configuration containing the statement "$x = 3 \,/\, (x + 2)\,;$" and state "$x \mapsto 1, y \mapsto 0$".

The state molecules and implicitly the state solution are straightforward. State molecules are not nested and state solutions are simply multisets of molecules of the form $x \mapsto i$. The CHAM does not allow us to impose constraints on solutions, such as that the molecules inside the state solution indeed define a partial function and not some arbitrary relation (i.e., there is at most one molecule $x \mapsto i$ for each $x \in \mathit{Id}$). Instead, the state solution will be used in such a way that the original state solution will embed a proper partial function and each rule will preserve this property. For example, the CHAM rule for variable assignment, say when assigning integer i to variable x, will rewrite the state molecule from $\{\!\{x \mapsto j \triangleleft \sigma\}\!\}$ to $\{\!\{x \mapsto i \triangleleft \sigma\}\!\}$.

The top level syntactic solution holds the current program or fragment of program that is still left to be processed. It is not immediately clear how the syntactic solution should be represented in order to be able to give IMP a CHAM semantics. The challenge here is that the IMP language constructs have evaluation strategies and the subterms that need to be next processed can be arbitrarily deep into the program or fragment of program, such as the framed x in "$x = 3 \,/\, (\boxed{x} + 2)\,;$". If the CHAM allowed conditional rules, then we could have followed an approach similar to that of SOS described in Section 3.3, reducing the semantics of each language construct to that of its subexpressions or substatements. Similarly, if the CHAM allowed matching using evaluation contexts, then we could have followed a reduction semantics with evaluation contexts approach like the one in Section 3.7.1 which uses only unconditional rules. Unfortunately, the CHAM allows neither conditional rules nor evaluation contexts in matching, so a different approach is needed.

Failed attempts to represent syntax. A natural approach to represent the syntax of a programming language in CHAM may be to try to use CHAM's heating/cooling and molecule/solution nesting mechanisms to decompose syntax unambiguously in such a way that the redex (i.e., the subterm which can be potentially reduced next; see Section 3.7) appears as a molecule in the top syntactic solution. That is, if $p = c[t]$ is a program or fragment of program which can be decomposed in evaluation context c and redex t, then one may attempt to represent it as a solution of the form $\{\!\{t\ \gamma_c\}\!\}$, where γ_c is some CHAM representation of the evaluation context c. If this worked, then we could use an airlock operation to isolate that redex from the

rest of the syntactic solution, i.e. $\{\!|p|\!\} \rightleftharpoons \{\!|t\ \gamma_c|\!\} \rightleftharpoons \{\!|t \triangleleft \{\!|\gamma_c|\!\}|\!\}$, and thus have it at the same level with the state solution in the configuration solution; this would allow to have rules that match both a syntactic molecule and a state molecule (after an airlock operation is applied on the state solution as well) in the same rule, as needed for the semantics of lookup and assignment. In our example above, we would obtain

$$\{\!|\{\!|x = 3\ /\ (x + 2)\,;|\!\}\ \{\!|x \mapsto 1\ \ y \mapsto 0|\!\}|\!\} \rightleftharpoons \{\!|\{\!|x \triangleleft \{\!|\gamma_{x=3/(\square+2)\,;}|\!\}|\!\}\ \{\!|x \mapsto 1 \triangleleft \{\!|y \mapsto 0|\!\}|\!\}|\!\}$$

and the latter could be rewritten with a natural CHAM reaction rule for variable lookup such as

$$\{\!|x \triangleleft c|\!\}\ \{\!|x \mapsto i \triangleleft \sigma|\!\} \rightarrow \{\!|i \triangleleft c|\!\}\ \{\!|x \mapsto i \triangleleft \sigma|\!\}$$

Unfortunately, there seems to be no way to achieve such a desirable CHAM representation of syntax. We next attempt and fail to do it in two different ways, and then give an argument why such a representation is actually impossible.

Consider, again, the statement "$x = 3\ /\ (x + 2)\,;$". A naive approach to represent this statement term as a syntactic solution (by means of appropriate heating/cooling rules) is to flatten it into its redex, namely x, and into all its atomic evaluation subcontexts, that is, to represent it as the following solution:

$$\{\!|x\ \ (\square + 2)\ \ (3\ /\ \square)\ \ (x = \square\,;)|\!\}$$

Such a representation can be relatively easily achieved by adding heating/cooling pair rules that correspond to the evaluation strategies (or contexts) of the various language constructs. For example, we can add the following rules corresponding to the evaluation strategies of the assignment and the addition constructs (and two similar ones for the division construct):

$$\begin{array}{rcl} x = a\,; & \rightleftharpoons & a \triangleleft \{\!|x = \square\,;|\!\} \\ a_1 + a_2 & \rightleftharpoons & a_1 \triangleleft \{\!|\square + a_2|\!\} \\ a_1 + a_2 & \rightleftharpoons & a_2 \triangleleft \{\!|a_1 + \square|\!\} \end{array}$$

With such rules, one can now heat or cool syntax as desired, for example:

$$\begin{array}{rcll} \{\!|x = 3\ /\ (x + 2)\,;|\!\} & \rightleftharpoons & \{\!|(3\ /\ (x + 2)) \triangleleft \{\!|x = \square\,;|\!\}|\!\} & \text{(Reaction)} \\ & \rightleftharpoons & \{\!|(3\ /\ (x + 2))\ \ (x = \square\,;)|\!\} & \text{(Airlock)} \\ & \rightleftharpoons & \{\!|(x + 2)\ \ (3\ /\ \square)\ \ (x = \square\,;)|\!\} & \text{(Reaction, Chemical, Airlock)} \\ & \rightleftharpoons & \{\!|x\ \ (\square + 2)\ \ (3\ /\ \square)\ \ (x = \square\,;)|\!\} & \text{(Reaction, Chemical, Airlock)} \end{array}$$

Unfortunately this naive approach is ambiguous, because it cannot distinguish the above from the representation of, say, $x = (3\ /\ x) + 2\,;$. The problem here is that the precise structure of the evaluation context is "lost in translation", so the approach above does not work.

Let us attempt a second approach, namely to guarantee that there is precisely one hole $\square$ molecule in each syntactic subsolution by using the molecule/solution nesting mechanism available in CHAM. More precisely, let us try to unambiguously represent the statements "$x = 3\ /\ (x + 2)\,;$" and "$x = (3\ /\ x) + 2\,;$" as the following two distinct syntactic solutions:

$$\begin{array}{l} \{\!|x\ \ \{\!|(\square + 2)\ \ \{\!|(3\ /\ \square)\ \ \{\!|(x = \square\,;)|\!\}|\!\}|\!\}|\!\} \\ \{\!|x\ \ \{\!|(3\ /\ \square)\ \ \{\!|(\square + 2)\ \ \{\!|(x = \square\,;)|\!\}|\!\}|\!\}|\!\} \end{array}$$

To achieve this, we modify the heating/cooling rules above as follows:

$$\begin{array}{rcl} (x = a\,;) \triangleleft c & \rightleftharpoons & a \triangleleft \{\!|\{\!|(x = \square\,;) \triangleleft c|\!\}|\!\} \\ (a_1 + a_2) \triangleleft c & \rightleftharpoons & a_1 \triangleleft \{\!|\{\!|(\square + a_2) \triangleleft c|\!\}|\!\} \\ (a_1 + a_2) \triangleleft c & \rightleftharpoons & a_2 \triangleleft \{\!|\{\!|(a_1 + \square) \triangleleft c|\!\}|\!\} \end{array}$$

With these modified rules, one may now think that one can heat and cool syntax unambiguously:

$$\begin{array}{rcll}
\{\!|x = 3 \,/\, (x+2)\,;|\!\} & \rightleftharpoons & \{\!|(x = 3 \,/\, (x+2)\,;) \triangleleft \{\!|\cdot|\!\}|\!\} & \text{(Airlock)} \\
& \rightleftharpoons & \{\!|(3 \,/\, (x+2)) \triangleleft \{\!|\{\!|(x = \square\,;) \triangleleft \{\!|\cdot|\!\}|\!\}|\!\} & \text{(Reaction)} \\
& \rightleftharpoons & \{\!|(3 \,/\, (x+2)) \triangleleft \{\!|\{\!|x = \square\,;|\!\}|\!\} & \text{(Airlock, Membrane)} \\
& \rightleftharpoons & \{\!|(x+2) \triangleleft \{\!|\{\!|(3 \,/\, \square) \triangleleft \{\!|\{\!|x = \square\,;|\!\}|\!\}|\!\} & \text{(Reaction)} \\
& \rightleftharpoons & \{\!|(x+2) \triangleleft \{\!|\{\!|(3 \,/\, \square)\ \{\!|x = \square\,;|\!\}|\!\} & \text{(Airlock, Membrane)} \\
& \rightleftharpoons & \{\!|x \triangleleft \{\!|\{\!|(\square + 2) \triangleleft \{\!|\{\!|(3 \,/\, \square)\ \{\!|x = \square\,;|\!\}|\!\}|\!\}|\!\} & \text{(Reaction)} \\
& \rightleftharpoons & \{\!|x \triangleleft \{\!|\{\!|(\square + 2)\ \{\!|(3 \,/\, \square)\ \{\!|x = \square\,;|\!\}|\!\}|\!\} & \text{(Airlock, Membrane)} \\
& \rightleftharpoons & \{\!|x\ \{\!|(\square + 2)\ \{\!|(3 \,/\, \square)\ \{\!|x = \square\,;|\!\}|\!\}|\!\} & \text{(Airlock)}
\end{array}$$

Unfortunately, the above is not the only way one can heat the solution in question. For example, the following is also a possible derivation, showing that these heating/cooling rules are still problematic:

$$\begin{array}{rcll}
\{\!|x = 3 \,/\, (x+2)\,;|\!\} & \rightleftharpoons & \{\!|(x = 3 \,/\, (x+2)\,;) \triangleleft \{\!|\cdot|\!\}|\!\} & \text{(Airlock)} \\
& \rightleftharpoons & \{\!|(3 \,/\, (x+2)) \triangleleft \{\!|\{\!|(x = \square\,;) \triangleleft \{\!|\cdot|\!\}|\!\}|\!\} & \text{(Reaction)} \\
& \rightleftharpoons & \{\!|(3 \,/\, (x+2)) \triangleleft \{\!|\{\!|x = \square\,;|\!\}|\!\} & \text{(Airlock, Membrane)} \\
& \rightleftharpoons & \{\!|(3 \,/\, (x+2))\ \{\!|x = \square\,;|\!\}|\!\} & \text{(Airlock)} \\
& \rightleftharpoons & \{\!|(x+2)\ \{\!|3 \,/\, \square|\!\}\ \{\!|x = \square\,;|\!\}|\!\} & \text{(All four laws)} \\
& \rightleftharpoons & \{\!|x\ \{\!|\square + 2|\!\}\ \{\!|3 \,/\, \square|\!\}\ \{\!|x = \square\,;|\!\}|\!\} & \text{(All four laws)}
\end{array}$$

Indeed, one can similarly show that

$$\{\!|x = (3 \,/\, x) + 2\,;|\!\} \quad \rightleftharpoons \quad \{\!|x\ \{\!|\square + 2|\!\}\ \{\!|3 \,/\, \square|\!\}\ \{\!|x = \square\,;|\!\}|\!\}$$

Therefore, this second syntax representation attempt is also ambiguous.

We claim that it is impossible to devise heating/cooling rules in CHAM and representations $\gamma_\_$ of evaluation contexts with the property that

$$\{\!|c[t]|\!\} \rightleftharpoons \{\!|t\ \gamma_c|\!\} \quad \text{or, equivalently,} \quad \{\!|c[t]|\!\} \rightleftharpoons \{\!|t \triangleleft \{\!|\gamma_c|\!\}|\!\}$$

for any term t and any appropriate evaluation context c. Indeed, if that was possible, then the following derivation could be possible:

$$\begin{array}{rcll}
\{\!|x = 3 \,/\, (x+2)\,;|\!\} & \rightleftharpoons & \{\!|(3 \,/\, (x+2))\ \gamma_{x=\square;}|\!\} & \text{(hypothesis)} \\
& \rightleftharpoons & \{\!|(x+2)\ \gamma_{3/\square}\ \gamma_{x=\square;}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|x\ \gamma_{\square+2}\ \gamma_{3/\square}\ \gamma_{x=\square;}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|(3 \,/\, x)\ \gamma_{\square+2}\ \gamma_{x=\square;}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|((3 \,/\, x) + 2)\ \gamma_{x=\square;}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|x = (3 \,/\, x) + 2\,;|\!\} & \text{(hypothesis)}
\end{array}$$

This general impossibility result explains why both our representation attempts above failed, as well as why many other similar attempts are also expected to fail.

The morale of the exercise above is that one should be very careful when using CHAM's airlock, because in combination with the other CHAM laws it can yield unexpected behaviors. In particular, the Chemical Law makes it impossible to state that a term matches the entire contents of a solution molecule, so one should not rely on the fact that all the remaining contents of a solution is in the membrane following the airlock. In our heating/cooling rules above, for example

$$\begin{array}{rcl}
(x = a\,;) \triangleleft c & \rightleftharpoons & a \triangleleft \{\!|(x = \square\,;) \triangleleft c|\!\} \\
(a_1 + a_2) \triangleleft c & \rightleftharpoons & a_1 \triangleleft \{\!|(\square + a_2) \triangleleft c|\!\} \\
(a_1 + a_2) \triangleleft c & \rightleftharpoons & a_2 \triangleleft \{\!|(a_1 + \square) \triangleleft c|\!\}
\end{array}$$

$$
\begin{array}{rcl}
a_1 + a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square + a_2 \curvearrowright c \\
a_1 + a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright a_1 + \square \curvearrowright c \\
a_1 \,/\, a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square \,/\, a_2 \curvearrowright c \\
a_1 \,/\, a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright a_1 \,/\, \square \curvearrowright c \\
a_1 \mathtt{<=} a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square \mathtt{<=} a_2 \curvearrowright c \\
i_1 \mathtt{<=} a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright i_1 \mathtt{<=} \square \curvearrowright c \\
\mathtt{!}\, b \curvearrowright c & \rightleftharpoons & b \curvearrowright \mathtt{!}\, \square \curvearrowright c \\
b_1 \mathtt{\&\&} b_2 \curvearrowright c & \rightleftharpoons & b_1 \curvearrowright \square \mathtt{\&\&} b_2 \curvearrowright c \\
x = a\mathtt{;} \curvearrowright c & \rightleftharpoons & a \curvearrowright x = \square\mathtt{;} \curvearrowright c \\
s_1\ s_2 \curvearrowright c & \rightleftharpoons & s_1 \curvearrowright \square\ s_2 \curvearrowright c \\
s & \rightleftharpoons & s \curvearrowright \square \\
\mathtt{if}\,(b)\ s_1\ \mathtt{else}\ s_2 \curvearrowright c & \rightleftharpoons & b \curvearrowright \mathtt{if}\,(\square)\ s_1\ \mathtt{else}\ s_2 \curvearrowright c
\end{array}
$$

Figure 3.44: CHAM heating-cooling rules for IMP.

our intuition that c matches *all* the evaluation context solution representation was wrong precisely for that reason. Indeed, it can just as well match a solution representation of a subcontext, which is why we got the unexpected derivation.

Correct representation of syntax. We next discuss an approach to representing syntax which is *not* based on CHAM's existing solution/membrane mechanism. We borrow from K (see Section 3.12) the idea of flattening syntax in an explicit list of computational tasks. Like in K, we use the symbol $\curvearrowright$, read "then" or "followed by", to separate such computational tasks; to avoid writing parentheses, we here assume that $\curvearrowright$ is right-associative and binds less tightly than any other construct. For example, the term "$x = 3 \,/\, (x + 2)\mathtt{;}$" gets represented as the list term

$$x \curvearrowright \square + 2 \curvearrowright 3 \,/\, \square \curvearrowright x = \square\mathtt{;} \curvearrowright \square$$

which reads "process x, followed by adding 2 to it, followed by dividing 3 by the result, followed by assigning the obtained result to x, which is the final task". Figure 3.44 shows all the heating/cooling rules that we associate to the various evaluation strategies of the IMP language constructs. These rules allow us to structurally rearrange any well-formed syntactic term so that the next computational task is at the top (left side) of the computation list.

The only rule in Figure 3.44 which does not correspond to the evaluation strategy of some evaluation construct is $s \rightleftharpoons s \curvearrowright \square$. Its role is to initiate the decomposition process whenever an unheated statement is detected in the syntax solution. According to CHAM's laws, these rules can only apply in solutions, so we can derive

$$\{\!|x = 1\mathtt{;}\ x = 3 \,/\, (x + 2)\mathtt{;}|\!\} \rightleftharpoons^* \{\!|x = 1\mathtt{;} \curvearrowright \square\ x = 3 \,/\, (x + 2)\mathtt{;} \curvearrowright \square|\!\}$$

but there is no way to derive, for example,

$$\{\!|x = 1\mathtt{;}\ x = 3 \,/\, (x + 2)\mathtt{;}|\!\} \rightleftharpoons^* \{\!|x = 1\mathtt{;}\ (x \curvearrowright \square{+}2 \curvearrowright 3 \,/\, \square \curvearrowright x = \square\mathtt{;} \curvearrowright \square)|\!\}$$

The syntactic solution will contain only one syntactic molecule at any given moment, with no subsolutions, which is the reason why the heating/cooling rules in Figure 3.44 that correspond to language construct evaluation strategies need to mention the remaining of the list of computational tasks in the syntactic molecule, c, instead of just the interesting part (e.g., $a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$, etc.), as we do in K (see

$$
\begin{array}{rcl}
\{\!|x \curvearrowright c|\!\}\,\{\!|x \mapsto i \triangleright \sigma|\!\} & \to & \{\!|i \curvearrowright c|\!\}\,\{\!|x \mapsto i \triangleright \sigma|\!\} \\
i_1 + i_2 \curvearrowright c & \to & i_1 +_{Int} i_2 \curvearrowright c \\
i_1 \,/\, i_2 \curvearrowright c & \to & i_1 /_{Int} i_2 \curvearrowright c \quad \text{when } i_2 \neq 0 \\
i_1 <= i_2 \curvearrowright c & \to & i_1 \leq_{Int} i_2 \curvearrowright c \\
\texttt{!}\,\texttt{true} \curvearrowright c & \to & \texttt{false} \curvearrowright c \\
\texttt{!}\,\texttt{false} \curvearrowright c & \to & \texttt{true} \curvearrowright c \\
\texttt{true}\,\texttt{\&\&}\,b_2 \curvearrowright c & \to & b_2 \curvearrowright c \\
\texttt{false}\,\texttt{\&\&}\,b_2 \curvearrowright c & \to & \texttt{false} \curvearrowright c \\
\{\,s\,\} \curvearrowright c & \to & s \curvearrowright c \\
\{\!|x = i\texttt{;} \curvearrowright c|\!\}\,\{\!|x \mapsto j \triangleright \sigma|\!\} & \to & \{\!|\{\} \curvearrowright c|\!\}\,\{\!|x \mapsto i \triangleright \sigma|\!\} \\
\{\}\ s_2 \curvearrowright c & \to & s_2 \curvearrowright c \\
\texttt{if}\,\texttt{(true)}\,s_1\,\texttt{else}\,s_2 \curvearrowright c & \to & s_1 \curvearrowright c \\
\texttt{if}\,\texttt{(false)}\,s_1\,\texttt{else}\,s_2 \curvearrowright c & \to & s_2 \curvearrowright c \\
\texttt{while}\,\texttt{(}b\texttt{)}\,s \curvearrowright c & \to & \texttt{if}\,\texttt{(}b\texttt{)}\,\{\,s\,\texttt{while}\,\texttt{(}b\texttt{)}\,s\,\}\,\texttt{else}\,\{\} \curvearrowright c \\
\texttt{int}\,xl\texttt{;}\ s & \to & \{\!|s|\!\}\,\{\!|xl \mapsto 0|\!\} \\
 & & \\
(x, xl) \mapsto i & \rightharpoonup & x \mapsto i \triangleright \{\!|xl \mapsto i|\!\}
\end{array}
$$

Figure 3.45: CHAM(IMP): The CHAM of IMP, obtained by adding to the heating/cooling rules in Figure 3.44 the semantic rules for IMP plus the heating rule for state initialization above.

Section 3.12). The heating/cooling rules in Figure 3.44 decompose the syntactic term into any of its possible splits into a redex (the top of the resulting list of computational tasks) and an evaluation context (represented flattened as the rest of the list). In fact, these heating/cooling rules have been almost mechanically derived from the syntax of evaluation contexts for the IMP language constructs in Section 3.7.1 (see Figure 3.30).

Figure 3.45 shows the remaining CHAM rules of IMP, giving the actual semantics of each language construct. Like the heating/cooling rules in Figure 3.44, these rules are also almost mechanically derived from the rules of the reduction semantics with evaluation contexts of IMP in Section 3.7.1 (see Figure 3.31), with the following notable differences:

- Each rule needs to mention the remaining list of computational tasks, c, for the same reason the heating/cooling rules in Figure 3.44 need to mention it (which is explained above).

- There is no equivalent of the characteristic rule of reduction semantics with evaluation contexts. The Membrane Law looks somehow similar, but we cannot take advantage of that because we were not able to use the inherent airlock mechanism of the CHAM to represent syntax (see the failed attempts to represent syntax above).

- The state is organized as a solution using CHAM's airlock mechanism, instead of just imported as an external data-structure as we did in our previous semantics. We did so because the state data-structure that we used in our previous semantics was a finite-domain partial function (see Section 3.1.2), which was represented as a set of pairs (Section 2.1.2), and it is quite natural to replace any set structures by the inherent CHAM solution mechanism.

Figure 3.46 shows a possible execution of IMP's CHAM defined above, mentioning at each step which of CHAM's laws have been applied. Note that the final configuration contains two subsolutions, one for the

$$
\begin{array}{rll}
\{\!|\texttt{int}\ x, y;\ x = 1;\ x = 3 \,/\, (x + 2);|\!\} & \to & \text{(Reaction)} \\
\{\!\{|x = 1;\ x = 3 \,/\, (x + 2);|\!\}\ \{|x, y \mapsto 0|\}\} & \rightharpoonup & \text{(Heating, Membrane)} \\
\{\!\{|x = 1;\ x = 3 \,/\, (x + 2);\ \curvearrowright \Box|\!\}\ \{|x, y \mapsto 0|\}\} & \rightharpoonup^* & \text{(Heating, Membrane, Airlock)} \\
\{\!\{|x = 1;\ x = 3 \,/\, (x + 2);\ \curvearrowright \Box|\!\}\ \{|x \mapsto 0\ \ y \mapsto 0|\}\} & \rightharpoonup & \text{(Heating, Membrane)} \\
\{\!\{|x = 1;\ \curvearrowright \Box\ x = 3 \,/\, (x + 2);\ \curvearrowright \Box|\!\}\ \{|x \mapsto 0\ \ y \mapsto 0|\}\} & \rightharpoonup & \text{(Airlock, Membrane)} \\
\{\!\{|x = 1;\ \curvearrowright \Box\ x = 3 \,/\, (x + 2);\ \curvearrowright \Box|\!\}\ \{|x \mapsto 0 \triangleright \{|y \mapsto 0|\}|\}\} & \to & \text{(Reaction)} \\
\{\!\{|\{\}\ \curvearrowright \Box\ x = 3 \,/\, (x + 2);\ \curvearrowright \Box|\!\}\ \{|x \mapsto 1 \triangleright \{|y \mapsto 0|\}|\}\} & \rightharpoonup & \text{(Cooling, Membrane)} \\
\{\!\{|\{\}\ x = 3 \,/\, (x + 2);\ \curvearrowright \Box|\!\}\ \{|x \mapsto 1 \triangleright \{|y \mapsto 0|\}|\}\} & \rightharpoonup & \text{(Reaction, Membrane)} \\
\{\!\{|x = 3 \,/\, (x + 2);\ \curvearrowright \Box|\!\}\ \{|x \mapsto 1 \triangleright \{|y \mapsto 0|\}|\}\} & \rightharpoonup^* & \text{(Heating, Membrane)} \\
\{\!\{|x \curvearrowright \Box + 2 \curvearrowright 3 \,/\, \Box \curvearrowright x = \Box;\ \curvearrowright \Box|\!\}\ \{|x \mapsto 1 \triangleright \{|y \mapsto 0|\}|\}\} & \to & \text{(Reaction)} \\
\{\!\{|1 \curvearrowright \Box + 2 \curvearrowright 3 \,/\, \Box \curvearrowright x = \Box;\ \curvearrowright \Box|\!\}\ \{|x \mapsto 1 \triangleright \{|y \mapsto 0|\}|\}\} & \rightharpoonup & \text{(Cooling)} \\
\{\!\{|1 + 2 \curvearrowright 3 \,/\, \Box \curvearrowright x = \Box;\ \curvearrowright \Box|\!\}\ \{|x \mapsto 1 \triangleright \{|y \mapsto 0|\}|\}\} & \to & \text{(Reaction, Membrane)} \\
\{\!\{|3 \curvearrowright 3 \,/\, \Box \curvearrowright x = \Box;\ \curvearrowright \Box|\!\}\ \{|x \mapsto 1 \triangleright \{|y \mapsto 0|\}|\}\} & \to^* & \text{(Reaction, Cooling, Membrane)} \\
\{\!\{|x = 1;\ \curvearrowright \Box|\!\}\ \{|x \mapsto 1 \triangleright \{|y \mapsto 0|\}|\}\} & \to & \text{(Reaction)} \\
\{\!\{|\{\}\ \curvearrowright \Box|\!\}\ \{|x \mapsto 1 \triangleright \{|y \mapsto 0|\}|\}\} & \to & \text{(Cooling, Membrane)} \\
\{\!\{|\{\}|\!\}\ \{|x \mapsto 1 \triangleright \{|y \mapsto 0|\}|\}\} & &
\end{array}
$$

Figure 3.46: An execution of IMP's CHAM.

syntax and one for the state. Since the syntactic subsolution in the final configuration solution is expected to always contain only {}, one can safely eliminated it.

3.8.2 The CHAM of IMP++

We next discuss the CHAM of IMP++, discussing like in the other semantics each feature separately first and then putting all of them together. When putting them together, we also investigate the modularity and appropriateness of the resulting definition.

Variable Increment

The chemical abstract machine can also define the increment modularly:

$$
\{\!|{+}{+}\ x \curvearrowright c|\!\}\ \{|x \mapsto i \triangleright \sigma|\} \to \{\!|i +_{Int} 1 \curvearrowright c|\!\}\ \{|x \mapsto i +_{Int} 1 \triangleright \sigma|\} \qquad \text{(CHAM-Inc)}
$$

Input/Output

All we have to do is to add new molecules in the top-level solution that hold the input and the output buffers, then define the evaluation strategy of `print` by means of a heating/cooling pair like we did for other strict constructs, and finally add the reaction rules corresponding to the input/output constructs. Since solutions are not typed, to distinguish the solution holding the input buffer from the one holding the output buffer, we introduce two artificial molecules, called input and output, respectively, and place them upfront in their corresponding solutions. Since the input buffer needs to also be provided within the initial solution, we modify the reaction rule for programs to take program and input molecules and initialize the output and state molecules accordingly:

$\texttt{print}(a)\texttt{;} \curvearrowright c \rightleftharpoons a \curvearrowright \texttt{print}(\square)\texttt{;} \curvearrowright c$

$\{\!|\texttt{read()} \curvearrowright c|\!\}\ \{\!|\textsf{input}\ i : w|\!\} \rightarrow \{\!|\{\} \curvearrowright c|\!\}\ \{\!|\textsf{input}\ w|\!\}$ (CHAM-Read)

$\{\!|\texttt{print}(i)\texttt{;} \curvearrowright c|\!\}\ \{\!|\textsf{output}\ w|\!\} \rightarrow \{\!|\{\} \curvearrowright c|\!\}\ \{\!|\textsf{output}\ w : i|\!\}$ (CHAM-Print)

$\{\!|\texttt{int}\ xl\texttt{;}\ s|\!\}\ \{\!|w|\!\} \rightarrow \{\!|s|\!\}\ \{\!|xl \mapsto 0|\!\}\ \{\!|\textsf{input}\ w|\!\}\ \{\!|\textsf{output}\ \epsilon|\!\}$ (CHAM-Pgm)

Abrupt Termination

The CHAM semantics of abrupt termination is even more elegant and modular than that using evaluation contexts above, because the other components of the configuration need not be mentioned:

$i\ /\ 0 \curvearrowright c \ \rightarrow\ \{\}$ (CHAM-Div-By-Zero)

$\texttt{halt;} \curvearrowright c \ \rightarrow\ \{\}$ (CHAM-Halt)

Dynamic Threads

As stated in Section 3.8, the CHAM has been specifically proposed as a model of concurrent computation, based on the chemical metaphor that molecules in solutions can get together and react, with possibly many reactions taking place concurrently. Since there was no concurrency so far in our language, the actual strength of the CHAM has not been seen yet. Recall that the configuration of the existing CHAM semantics of IMP consists of one top-level solution, which contains two subsolutions: a syntactic subsolution holding the remainder of the program organized as a molecule sequentializing computation tasks using the special construct $\curvearrowright$; and a state subsolution containing binding molecules, each binding a different program variable to a value. As seen in Figures 3.44 and 3.45, most of the CHAM rules involve only the syntactic molecule. The state subsolution is only mentioned when the language construct involves program variables.

The above suggests that all a `spawn` statement needs to do is to create an additional syntactic subsolution holding the spawned statement, letting the newly created subsolution molecule to float together with the original syntactic molecule in the same top-level solution. Minimalistically, this can be achieved with the following CHAM rule (which does not consider thread termination yet):

$$\{\!|\texttt{spawn}\ s \curvearrowright c|\!\} \rightarrow \{\!|\{\} \curvearrowright c|\!\}\ \{\!|s|\!\}$$

Since the order of molecules in a solution is irrelevant, the newly created syntactic molecule has the same rights as the original molecule in reactions involving the state molecule. We can (correctly) think of each syntactic subsolution as an independently running execution thread. The same CHAM rules we had before (see Figures 3.44 and 3.45) can now also apply to the newly created threads. Moreover, reactions taking place only in the syntactic molecules, which are a majority by a large number, can apply truly concurrently. For example, a thread may execute a loop unrolling step while another thread may concurrently perform an addition. The only restriction regarding concurrency is that rule instances must involve disjoint molecules in order to proceed concurrently. That means that it is also possible for a thread to read or write the state while another thread, truly concurrently, performs a local computation. This degree of concurrency was not possible within the other semantic approaches discussed so far in this chapter.

The rule above only creates threads. It does not collect threads when they complete their computation. One could do that with the simple solution-dissolving rule

$$\{\!|\{\}|\!\} \rightarrow \cdot$$

but the problem is that such a rule cannot distinguish between the original thread and the others, so it would also dissolve the original thread when it completes. This could be considered correct behavior, but, however,

we prefer to distinguish the original thread created statically from the others, which are created dynamically. Specifically, we collect only the terminated threads which were created dynamically. To achieve that, we can flag the newly created threads for collection as below. Here is our complete CHAM semantics of `spawn`:

$$\textit{Molecule} \;::=\; \ldots \mid \texttt{die}$$

$$\{\!|\texttt{spawn}\ s \curvearrowright c|\!\} \rightarrow \{\!|\{\} \curvearrowright c|\!\}\ \{\!|s \curvearrowright \texttt{die}|\!\} \qquad \text{(CHAM-Spawn)}$$

$$\{\!|\{\} \curvearrowright \texttt{die}|\!\} \rightarrow \cdot \qquad \text{(CHAM-Die)}$$

We conclude this section with a discussion on the concurrency of the CHAM above. As already argued, it allows for truly concurrent computations to take place, provided that their corresponding CHAM rule instances do not overlap. While this already goes far beyond the other semantical approaches in terms of concurrency, it still enforces interleaving where it should not.

Consider, for example, a global configuration in which two threads are about to lookup two different variables in the state cell. Even though there are good reasons to allow the two threads to proceed concurrently, the CHAM above will not, because the two rule instances (of the same CHAM lookup rule) overlap on the state molecule. This problem can be ameliorated, to some extent, by changing the structure of the top-level configuration to allow all the variable binding molecules currently in the state subsolution to instead float in the top-level solution at the same level with the threads: this way, each thread can independently grab the binding it is interested in without blocking the state anymore. Unfortunately, this still does not completely solve the true concurrency problem, because one could argue that different threads should also be allowed to concurrently read the same variable. Thus, no matter where the binding of that variable is located, the two rule instances cannot proceed concurrently. Moreover, flattening all the syntactic and the semantic ingredients in a top level solution, as the above "fix" suggests, does not scale. Real-life languages can have many configuration items of various kinds, such as, environments, heaps, function/exception/loop stacks, locks held, and so on. Collapsing the contents of all these items in one flat solution would not only go against the CHAM philosophy, but it would also make it hard to understand and control. The K framework (see Section 3.12) solves this problem by allowing its rules to state which parts of the matched subterm are shared with and which can be concurrently modified by other rules.

Local Variables

The simplest approach to adding blocks with local variables to IMP is to follow an idea similar to the one for reduction semantics with evaluation contexts discussed in Section 3.7.2, assuming the procedure presented in Section 3.5.5 for desugaring blocks with local variables into `let` constructs:

$$\texttt{let}\ x = a\ \texttt{in}\ s \curvearrowright c \rightleftharpoons a \curvearrowright \texttt{let}\ x = \square\ \texttt{in}\ s \curvearrowright c$$
$$\{\!|\texttt{let}\ x = i\ \texttt{in}\ s \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \rightarrow \{\!|s\ x = \sigma(x);\ \curvearrowright c|\!\}\ \{\!|\sigma[i/x]|\!\} \qquad \text{(CHAM-Let)}$$

As it was the case in Section 3.7.2, the above is going to be problematic when we add `spawn` to the language, too. However, recall that in this language experiment we pretend each language extension is final, in order to understand the modularity and flexibility to change of each semantic approach.

The approach above is very syntactic in nature, following the intuitions of evaluation contexts. In some sense, the above worked because we happened to have an assignment statement in our language, which we used for recovering the value of the bound variable. Note, however, that several computational steps were wasted because of the syntactic translations. What we would have really liked to say is "`let` *Id* = *Int* `in` *Context* is a special evaluation context where the current state is updated with the binding

whenever `let` is passed through top-down, and where the state is recovered whenever `let` is passed through bottom-up". This was not possible to say with evaluation contexts. When using the CHAM, we are free to disobey the syntax. For example, the alternative definition below captures the essence of the problem and wastes no steps (although it is still problematic when combined with `spawn`):

$$\texttt{let}\ x = a\ \texttt{in}\ s \curvearrowright c \rightleftharpoons a \curvearrowright \texttt{let}\ x = \square\ \texttt{in}\ s \curvearrowright c$$
$$\{\!|\texttt{let}\ x = i\ \texttt{in}\ s \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \rightharpoonup \{\!|s \curvearrowright \texttt{let}\ x = \sigma(x)\ \texttt{in}\ \square \curvearrowright c|\!\}\ \{\!|\sigma[i/x]|\!\}$$
$$\{\!|\{\} \curvearrowright \texttt{let}\ x = v\ \texttt{in}\ \square \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \rightharpoonup \{\!|\{\} \curvearrowright c|\!\}\ \{\!|\sigma[v/x]|\!\}$$

In words, the `let` is first heated/cooled in the binding expression. Once that becomes an integer, the `let` is then only heated in its body statement, at the same time updating the state molecule with the binding and storing the return value in the residual `let` construct. Once the `let` body statement becomes {}, the solution is cooled down by discarding the residual `let` construct and recovering the state appropriately (we used a "value" v instead of an integer i in the latter rule to indicate the fact that v can also be $\bot$).

Of course, the substitution-based approach discussed in detail in Sections 3.5.6 and 3.7.2 can also be adopted here if one is willing to pay the price for using it:

$$\texttt{let}\ x = a\ \texttt{in}\ s \curvearrowright c \rightleftharpoons a \curvearrowright \texttt{let}\ x = \square\ \texttt{in}\ s \curvearrowright c$$
$$\{\!|\texttt{let}\ x = i\ \texttt{in}\ s \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \rightarrow \{\!|s[x'/x] \curvearrowright c|\!\}\ \{\!|\sigma[i/x']|\!\} \quad \text{if } x' \text{ is a fresh variable}$$

Putting Them All Together

Putting together all the language features defined in CHAM above is a bit simpler and more modular than in MSOS (see Section 3.6.2): all we have to do is to take the union of all the syntax and semantics of all the features, removing the original rule for the initialization of the solution (that rule was already removed as part of the addition of input/output to IMP); in MSOS, we also had to add the halting attribute to the labels, which we do not have to do in the case of the CHAM.

Unfortunately, like in the case of the reduction semantics with evaluation contexts of IMP++ in Section 3.7.2, the resulting language is flawed. Indeed, a thread spawned from inside a `let` would be created its own molecule in the top-level solution, which would execute concurrently with all the other execution threads, including its parent. Thus, there is the possibility that the parent will advance to the assignment recovering the value of the `let`-bound variable before the spawned thread terminates, in which case the bound variable would be changed in the state by the parent thread, "unexpectedly" for the spawned thread. One way to address this problem is to rename the bound variable into a fresh variable within the `let` body statement, like we did above, using a substitution operation. Another is to split the state into an environment mapping variables to locations and a store mapping locations to values, and to have each thread consist of a solution holding both its code and its environment. Both these solutions were also suggested in Section 3.5.6, when we discussed how to make small-step SOS correctly capture all the behaviors of the resulting IMP++.

3.8.3 CHAM in Rewrite Logic

As explained above, CHAM rewriting cannot be immediately captured as ordinary rewriting modulo solution multiset axioms such as associativity, commutativity and identity. The distinction between the two arises essentially when a CHAM rule involves only one top-level molecule which is not a solution, because the CHAM laws restrict the applications of such a rule only in solutions while ordinary rewriting allows such rules to apply everywhere. To solve this problem, we wrap each rule in a solution context, that is, we translate CHAM rules of the form

$$m_1\ m_2\ \ldots\ m_k \rightarrow m'_1\ m'_2\ \ldots\ m'_l$$

into corresponding rewrite logic rules of the form

$$\{\!|\overline{m_1}\ \overline{m_2}\ \ldots\ \overline{m_k}\ Ms|\!\} \rightarrow \{\!|\overline{m'_1}\ \overline{m'_2}\ \ldots\ \overline{m'_l}\ Ms|\!\}$$

where the only difference between the original CHAM terms $m_1, m_2, \ldots, m_k, m'_1, m'_2, \ldots, m'_l$ and their algebraic variants $\overline{m_1}, \overline{m_2}, \ldots, \overline{m_k}, \overline{m'_1}, \overline{m'_2}, \ldots, \overline{m'_l}$ is that the meta-variables appearing in the former (recall that CHAM rules are schemata) are turned into variables of corresponding sorts in the latter, and where *Ms* is a variable of sort **Bag{***Molecule***}** that does not appear anywhere else in $\overline{m_1}, \overline{m_2}, \ldots, \overline{m_k}, \overline{m'_1}, \overline{m'_2}, \ldots, \overline{m'_l}$.

With this representation of CHAM rules into rewrite logic rules, it is obvious that rewrite logic's rewriting captures both the Reaction Law and the Chemical Law of the CHAM. What is less obvious is that it also captures the Membrane Law. Indeed, note that the Membrane Law allows rewrites to take place only when the global term is a solution, while rewriting logic allows rewrites to take place anywhere. However, the rewrite logic rule representation above generates only rules that rewrite solutions into solutions. Thus, if the original term to rewrite is a solution, then so it will stay during the entire rewriting process, and so the Membrane Law is also naturally captured by rewrite logic derivations. However, if the original term to rewrite is a proper molecule (which is not a solution), then so it will stay during the entire rewrite logic's rewriting while the CHAM will not reduce it at all. Still it is important to understand that in this case the corresponding rewrite logic theory can perform rewrite steps (in subsolutions of the original term) which are not possible under the CHAM, in particular that it may lead to non-termination in situations where the CHAM is essentially stuck. To reconcile this inherent difference between the CHAM and rewrite logic, we make the reasonable assumption that the original terms to rewrite can only be solutions. Note that the CHAM sequents in Definition 23 already assume that one only derives solution terms.

The only CHAM law which has not been addressed above is the Airlock Law. Rewrite logic has no builtin construct resembling CHAM's airlock, but its multiset matching is powerful enough to allow us to capture the airlock's behavior through rewrite rules. One possibility is to regard the airlock operation like any other molecular construct. Indeed, from a rewrite logic perspective, the Airlock Law says that any molecule inside a solution can be matched and put into an airlock next to the remaining solution wrapped into a membrane, and this process is reversible. This behavior can be achieved through the following two (opposite) rewrite logic rules, where *M* is a molecule variable and *Ms* is a bag-of-molecules variable:

$$\begin{array}{rcl} \{\!|M\ Ms|\!\} & \rightarrow & \{\!|M \rhd \{\!|Ms|\!\}|\!\} \\ \{\!|M \rhd \{\!|Ms|\!\}|\!\} & \rightarrow & \{\!|M\ Ms|\!\} \end{array}$$

Another possibility to capture airlock's behavior in rewrite logic is to attempt to eliminate it completely and replace it with matching modulo multiset axioms. While this appears to be possible in many concrete situations, we are however not aware of any general solution to do so systematically for any CHAM. The question is whether the elimination of the airlock is indeed safe, in the sense that the resulting rewrite logic theory does not lose any of the original CHAM's behaviors. One may think that thanks to the restricted form of CHAM's rules, the answer is immediately positive. Indeed, since the CHAM disallows any other constructs for solutions except its builtin membrane operation (a CHAM can only add new syntactic constructs for molecules, but not for solutions) and since solution subterms can either contain only one molecule or otherwise be meta-variables (to avoid multiset matching), we can conclude that in any CHAM rule, a subterm containing an airlock operation at its top can only be of the form $m \rhd \{\!|m'|\!\}$ or of the form $m \rhd s$ with s a meta-variable. Both these cases can be uniformly captured as subterms of the form $\overline{m} \rhd \{\!|ms|\!\}$ with *ms* a term of sort **Bag{***Molecule***}** in rewrite logic, the former by taking $k = 1$ and $ms = m'$ and the latter by replacing the metavariable s with a term of the form $\{\!|Ms|\!\}$ everywhere in the rule, where *Ms* is a fresh variable of sort **Bag{***Molecule***}**. Unfortunately, it is not clear how we can eliminate the airlock from subterms of the form

sorts:
Molecule, *Solution*, **Bag{***Molecule***}**
subsorts:
Solution < *Molecule*
// One may also need to subsort to *Molecule* specific syntactic categories (*Int*, *Bool*, etc.)
operations:
$\{\!|\_|\!\}$: **Bag{***Molecule***}** → *Solution* // membrane operator
$\_ \triangleright \_$: *Molecule* × *Solution* → *Molecule* // airlock operator
// One may also need to define specific syntactic constructs for *Molecule* ($\_ + \_$, $\_ \mapsto \_$, etc.)
rules:
// Add the following two generic (i.e., same for all CHAMs) airlock rewrite logic rules:
$\{\!|M\ Ms|\!\} \leftrightarrow \{\!|M \triangleright \{\!|Ms|\!\}|\!\}$ // M, Ms variables of sorts *Molecule*, **Bag{***Molecule***}**, resp.
// For each specific CHAM rule $m_1\ m_2\ \ldots m_k \rightarrow m'_1\ m'_2\ \ldots m'_l$ add a rewrite logic rule
$\{\!|\overline{m_1}\ \overline{m_2}\ \ldots\ \overline{m_k}\ Ms|\!\} \rightarrow \{\!|\overline{m'_1}\ \overline{m'_2}\ \ldots\ \overline{m'_l}\ Ms|\!\}$ // Ms variable of sort **Bag{***Molecule***}**
// where $\overline{m}$ replaces each meta-variable in m by a variable of corresponding sort.

Figure 3.47: Embedding of a chemical abstract machine into rewrite logic (CHAM $\rightsquigarrow \mathcal{R}_{\text{CHAM}}$).

$\overline{m} \triangleright \{\!|ms|\!\}$. If such terms appear in a solution or at the top of the rule, then one can replace them by their corresponding bag-of-molecule terms $\overline{m}\ ms$. However, if they appear in a proper molecule context (i.e., not in a solution context), then they cannot be replaced by $\overline{m}\ ms$ (first, we would get a parsing error by placing a bag-of-molecule term in a molecule place; second, reactions cannot take place in *ms* anymore, because it is not surrounded by a membrane). We cannot replace $\overline{m} \triangleright \{\!|ms|\!\}$ by $\{\!|\overline{m}\ ms|\!\}$ either, because that would cause a double membrane when the thus modified airlock reaches a solution context. Therefore, we keep the airlock.

Putting all the above together, we can associate a rewrite logic theory to any CHAM as shown in Figure 3.47; for simplicity, we assumed that the CHAM has only one *Molecule* syntactic category and, implicitly, only one corresponding *Solution* syntactic category. In Figure 3.47 and elsewhere in this section, we use the notation *left* ↔ *right* as a shorthand for two opposite rewrite rules, namely for both *left* → *right* and *right* → *left*. The discussion above implies the following result:

Theorem 21. ***(Embedding of the chemical abstract machine into rewrite logic)*** *If* CHAM *is a chemical abstract machine, sol and sol′ are two solutions, and* $\mathcal{R}_{\text{CHAM}}$ *is the rewrite logic theory associated to* CHAM *as in Figure 3.47, then the following hold:*

1. CHAM $\vdash sol \rightarrow sol'$ *if and only if* $\mathcal{R}_{\text{CHAM}} \vdash \overline{sol} \rightarrow^1 \overline{sol'}$;
2. CHAM $\vdash sol \rightarrow^* sol'$ *if and only if* $\mathcal{R}_{\text{CHAM}} \vdash \overline{sol} \rightarrow \overline{sol'}$.

Therefore, one-step solution rewriting in $\mathcal{R}_{\text{CHAM}}$ corresponds precisely to one-step solution rewriting in the original CHAM and thus, one can use $\mathcal{R}_{\text{CHAM}}$ as a replacement for CHAM for any reduction purpose. Unfortunately, this translation of CHAM into rewrite logic does not allow us to borrow the latter's concurrency to obtain the desired concurrent rewriting computational mechanism of the former. Indeed, the desired CHAM concurrency says that "different rule instances can apply concurrently in the same solution as far as they act on different molecules". Unfortunately, the corresponding rewrite logic rule instances cannot apply concurrently according to rewrite logic's semantics because both instances would match the entire solution, including the membrane (and rule instances which overlap cannot proceed concurrently in rewrite logic—see Section 2.5).

The CHAM of IMP in Rewrite Logic

Figure 3.48 shows the rewrite theory $\mathcal{R}_{\text{CHAM(IMP)}}$ obtained by applying the generic transformation procedure in Figure 3.47 to the CHAM of IMP discussed in this section and summarized in Figures 3.44 and 3.45. In addition to the generic CHAM syntax, as indicated in the comment under subsorts in Figure 3.47 we also subsort the builtin sorts of IMP, namely *Int*, *Bool* and *Id*, to *Molecule*. The "followed by" $\_ \curvearrowright \_$ construct for molecules is necessary for defining the evaluation strategies of the various IMP language constructs as explained above; we believe that some similar operator is necessary when defining any programming language whose constructs have evaluation strategies because, as explained above, it appears that the CHAM airlock operator is not suitable for this task. For notational simplicity, we stick to our previous convention that $\_ \curvearrowright \_$ is right associative and binds less tight than any other molecular construct. Finally, we add molecular constructs corresponding to all the syntax that we need in order to define the IMP semantics, which includes syntax for language constructs, for evaluation contexts, and for the state and state initialization.

The rewrite rules in Figure 3.48 are straightforward, following the transformation described in Figure 3.47. We grouped them in four categories: (1) the airlock rule is reversible and precisely captures the CHAM Airlock Law; (2) the heating/cooling rules, also reversible, capture the evaluation strategies of IMP's language constructs (see Figure 3.44); (3) the semantic rules are irreversible and capture the computational steps of the IMP semantics (see Figure 3.45); (4) the state initialization rule corresponds to the heating rule in Figure 3.45.

Our CHAM-based rewrite logic semantics of IMP in Figure 3.48 follows blindly the CHAM of IMP in Figures 3.44 and 3.45. All it does is to mechanically apply the transformation in Figure 3.47, without making any attempts to optimize the resulting rewrite logic theory. For example, it is easy to see that the syntactic molecules will always contain only one molecule. Also, it is easy to see that the state molecule can be initialized in such a way that at any moment during the initialization rewriting sequence any subsolution containing a molecule of the form $xl \mapsto i$, with xl a proper list, will contain no other molecule. Finally, one can also notice that the top level solution will always contain only two molecules, namely what we called a syntactic solution and a state solution. All these observations suggest that we can optimize the rewrite logic theory in Figure 3.48 by deleting the variable *Ms* from every rule except the airlock ones. While one can do that for our simple IMP language here, one has to be careful with such optimizations in general. For example, we later on add threads to IMP (see Section 3.5), which implies that the top level solution will contain a dynamic number of syntactic subsolutions, one per thread; if we remove the *Ms* variable from the lookup and assignment rules in Figure 3.48, then those rules will not work whenever there are more than two threads running concurrently. On the other hand, the *Ms* variable in the rule for variable declarations can still be eliminated. The point is that one needs to exercise care when one attempts to hand-optimize the rewrite logic theories resulting from mechanical semantic translations.

★ The CHAM of IMP in Maude

Like for the previous semantics, it is relatively straightforward to mechanically translate the corresponding rewrite theories into Maude modules. However, unlike in the previous semantics, the resulting Maude modules are not immediately executable. The main problem is that, in spite of its elegant chemical metaphor, the CHAM was not conceived to be blindly executable. For example, most of the heating and cooling rules tend to be reversible, leading to non-termination of the underlying rewrite relation. Non-termination is not a problem per se in rewrite logic and in Maude, because one can still use other formal analysis capabilities of these such as search and model checking, but from a purely pragmatic perspective it is rather inconvenient not to be able to execute an operational semantics of a language, particularly of a simple one like our IMP. Moreover, since the state space of a CHAM can very quickly grow to unmanageable sizes even when the

sorts:

$\mathit{Molecule}$, $\mathit{Solution}$, $\mathbf{Bag}\{\mathit{Molecule}\}$ // generic CHAM sorts

subsorts:

$\mathit{Solution} < \mathit{Molecule}$ // generic CHAM subsort

Int, Bool, $\mathit{Id} < \mathit{Molecule}$ // additional IMP-specific syntactic categories

operations:

$\{\!|\_|\!\} : \mathbf{Bag}\{\mathit{Molecule}\} \to \mathit{Solution}$ // generic membrane

$\_ \triangleright \_ : \mathit{Molecule} \times \mathit{Solution} \to \mathit{Molecule}$ // generic airlock

$\_ \curvearrowright \_ : \mathit{Molecule} \times \mathit{Molecule} \to \mathit{Molecule}$// "followed by" operator, for evaluation strategies

// Plus all the IMP language constructs and evaluation contexts (all listed in Figure 3.30),

// collapsing all syntactic categories different from Int, Bool and Id into $\mathit{Molecule}$

rules:

// Airlock:

$\{\!|M\ Ms|\!\} \leftrightarrow \{\!|M \triangleright \{\!|Ms|\!\}|\!\}$

// Heating/cooling rules corresponding to the evaluation strategies of IMP's constructs:

$\{\!|(A_1 + A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_1 \curvearrowright \square + A_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 + A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_2 \curvearrowright A_1 + \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 \,/\, A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_1 \curvearrowright \square \,/\, A_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 \,/\, A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_2 \curvearrowright A_1 \,/\, \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 \mathrel{<=} A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_1 \curvearrowright \square \mathrel{<=} A_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(I_1 \mathrel{<=} A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_2 \curvearrowright I_1 \mathrel{<=} \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{!}\ B \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(B \curvearrowright \texttt{!}\ \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(B_1 \mathbin{\&\&} B2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(B_1 \curvearrowright \square \mathbin{\&\&} B_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(X = A\texttt{;} \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A = \square\texttt{;} \curvearrowright C)\ Ms|\!\}$

$\{\!|(S_1\ S_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(S_1 \curvearrowright \square\ S_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|S\ Ms|\!\} \leftrightarrow \{\!|(S \curvearrowright \square)\ Ms|\!\}$

$\{\!|(\texttt{if (}B\texttt{)}\ S_1\ \texttt{else}\ S_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(B \curvearrowright \texttt{if (}\square\texttt{)}\ S_1\ \texttt{else}\ S_2 \curvearrowright C)\ Ms|\!\}$

// Semantic rewrite rules corresponding to reaction computational steps

$\{\!|\{\!|X \curvearrowright C|\!\}\ \{\!|X \mapsto I \triangleright \sigma|\!\}\ Ms|\!\} \to \{\!|\{\!|I \curvearrowright C|\!\}\ \{\!|X \mapsto I \triangleright \sigma|\!\}\ Ms|\!\}$

$\{\!|(I_1 + I_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(I_1 +_{\mathit{Int}} I_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(I_1 \,/\, I_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(I_1 /_{\mathit{Int}} I_2 \curvearrowright C)\ Ms|\!\}$ **if** $I_2 \neq 0$

$\{\!|(I_1 \mathrel{<=} I_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(I_1 \leq_{\mathit{Int}} I_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{!}\ \texttt{true} \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{false} \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{!}\ \texttt{false} \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{true} \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{true} \mathbin{\&\&} B2 \curvearrowright C)\ Ms|\!\} \to \{\!|(B_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{false} \mathbin{\&\&} B2 \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{false} \curvearrowright C)\ Ms|\!\}$

$\{\!|(\{\ S\ \} \curvearrowright C)\ Ms|\!\} \to \{\!|(S \curvearrowright C)\ Ms|\!\}$

$\{\!|\{\!|X = I\texttt{;} \curvearrowright C|\!\}\ \{\!|X \mapsto J \triangleright \sigma|\!\}\ Ms|\!\} \to \{\!|\{\!|\{\}\ \curvearrowright C|\!\}\ \{\!|X \mapsto I \triangleright \sigma|\!\}\ Ms|\!\}$

$\{\!|(\{\}\ S_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(S_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{if (true)}\ S_1\ \texttt{else}\ S_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(S_1 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{if (false)}\ S_1\ \texttt{else}\ S_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(S_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{while (}B\texttt{)}\ S \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{if (}B\texttt{)}\ \{\ S\ \texttt{while (}B\texttt{)}\ S\ \}\ \texttt{else}\ \{\} \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{int}\ xl\texttt{;}\ s)\ Ms|\!\} \to \{\!|\{\!|s|\!\}\ \{\!|xl \mapsto 0|\!\}\ Ms|\!\}$

// State initialization:

$\{\!|(X, Xl \mapsto I)\ Ms|\!\} \to \{\!|(X \mapsto I \triangleright \{\!|Xl \mapsto I|\!\})\ Ms|\!\}$

Figure 3.48: $\mathcal{R}_{\text{CHAM(IMP)}}$: The CHAM of IMP in rewrite logic.

```
mod CHAM is
  sorts Molecule Solution Bag{Molecule} .
  subsort Solution < Molecule < Bag{Molecule} .
  op empty : -> Bag{Molecule} .
  op _#_ : Bag{Molecule} Bag{Molecule} -> Bag{Molecule} [assoc comm id: empty] .
  op {|_|} : Bag{Molecule} -> Solution .
  op _<|_ : Molecule Solution -> Molecule [prec 110] .
  var M : Molecule .  var Ms : Bag{Molecule} .
  rl {| M # Ms |} => {| M <| {| Ms |} |} .
  rl {| M <| {| Ms |} |} => {| M # Ms |} .
endm

mod IMP-CHAM-SYNTAX is including PL-INT + CHAM .
  subsort Int < Molecule .
--- Define all the IMP constructs as molecule constructs
  op _+_ : Molecule Molecule -> Molecule [prec 33 gather (E e) format (d b o d)] .
  op _/_ : Molecule Molecule -> Molecule [prec 31 gather (E e) format (d b o d)] .
--- ... and so on
--- Add the hole as basic molecular construct, to allow for building contexts as molecules
  op [] : -> Molecule .
endm

mod IMP-HEATING-COOLING-CHAM-FAILED-1 is including IMP-CHAM-SYNTAX .
  var A1 A2 : Molecule .  var Ms : Bag{Molecule} .
--- + strict in its first argument
  rl {| (A1 + A2) # Ms |} => {| (A1 <| {| [] + A2 |}) # Ms |} .
  rl {| (A1 <| {| [] + A2 |}) # Ms |} => {| (A1 + A2) # Ms |} .
--- / strict in its second argument
  rl {| (A1 / A2) # Ms |} => {| (A2 <| {| A1 / [] |}) # Ms |} .
  rl {| (A2 <| {| A1 / [] |}) # Ms |} => {| (A1 / A2) # Ms |} .
--- ... and so on
endm

mod IMP-HEATING-COOLING-CHAM-FAILED-2 is including IMP-CHAM-SYNTAX .
  var A1 A2 : Molecule .  var Ms : Bag{Molecule} .  var C : Solution .
--- + strict in its first argument
  rl {| (A1 + A2 <| C) # Ms |} => {| (A1 <| {| {| [] + A2 <| C |} |}) # Ms |} .
  rl {| (A1 <| {| {| [] + A2 <| C |} |}) # Ms |} => {| (A1 + A2 <| C) # Ms |} .
--- / strict in its second argument
  rl {| (A1 / A2 <| C) # Ms |} => {| (A2 <| {| {| A1 / [] <| C |} |}) # Ms |} .
  rl {| (A2 <| {| {| A1 / [] <| C |} |}) # Ms |} => {| (A1 / A2 <| C) # Ms |} .
--- ... and so on
endm
```

Figure 3.49: Failed attempts to represent the CHAM of IMP in Maude using the airlock mechanism to define evaluation strategies. This figure also highlights the inconvenience of redefining IMP's syntax (the module `IMP-CHAM-SYNTAX` needs to redefine the IMP syntax as molecule constructs).

state space of the represented program is quite small, a direct representation of a CHAM in Maude can easily end up having only a theoretical relevance.

Before addressing the non-termination issue, it is instructive to discuss how a tool like Maude can help us pinpoint and highlight potential problems in our definitions. For example, we have previously seen how we failed, in two different ways, to use CHAM's airlock operator to define the evaluation strategies of the various IMP language constructs. We have noticed those problems with using the airlock for evaluation strategies by actually experimenting with CHAM definitions in Maude, more precisely by using Maude's search command to explore different behaviors of a program. We next discuss how one can use Maude to find out that both our attempts to use airlock for evaluation strategies fail. Figure 3.49 shows all the needed modules. `CHAM` defines the generic syntax of the chemical abstract machine together with its airlock rules in Maude, assuming only one type of molecule. `IMP-CHAM-SYNTAX` defines the syntax of IMP as well as the syntax of IMP's evaluation contexts as a syntax for molecules; since there is only one syntactic category for syntax now, namely `Molecule`, adding □ as a `Molecule` constant allows for `Molecule` to also include all the IMP evaluation contexts, as well as many other garbage terms (e.g., □ + □, etc.). The module `IMP-HEATING-COOLING-CHAM-FAILED-1` represents in Maude, using the general translation of CHAM rules into rewrite logic rules shown in Figure 3.47, heating/cooling rules of the form

$$a_1 + a_2 \;\rightleftharpoons\; a_1 \triangleleft \{\!|\, \square + a_2 \,|\!\}$$

Only two such groups of rules are shown, which is enough to show that we have a problem. Indeed, all four Maude search commands below succeed:

```
search[1] {| 3 / (1 + 2) |} =>* {| 1 ([] + 2) (3 / []) |} .
search[1] {| 1 ([] + 2) (3 / []) |} =>* {| 3 / (1 + 2) |} .
search[1] {| (3 / 1) + 2 |} =>* {| 1 ([] + 2) (3 / []) |} .
search[1] {| 1 ([] + 2) (3 / []) |} =>* {| (3 / 1) + 2 |} .
```

That means that the Maude solution terms `{| 3 / (1 + 2) |}` and `{| (3 / 1) + 2 |}` can rewrite into each other, which is clearly wrong. Similarly, `IMP-HEATING-COOLING-CHAM-FAILED-2` represents in Maude heating/cooling rules of the form

$$(a_1 + a_2) \triangleleft c \;\rightleftharpoons\; a_1 \triangleleft \{\!|\, (\square + a_2) \triangleleft c \,|\!\}$$

One can now check that this second approach gives us what we wanted, that is, a chemical representation of syntax where each subsolution directly contains no more than one □:

```
search[1] {| 3 / (1 + 2) |} =>* {| 1 {| ([] + 2) {| 3 / [] |} |} |} .
search[1] {| (3 / 1) + 2 |} =>* {| 1 {| (3 / []) {| [] + 2 |} |} |} .
```

Indeed, both Maude search commands above succeed. Unfortunately, the following commands

```
search[1] {| 3 / (1 + 2) |} =>* {| 1 {| [] + 2 |} {| 3 / [] |} |} .
search[1] {| 1 {| [] + 2 |} {| 3 / [] |} |} =>* {| 3 / (1 + 2) |} .
search[1] {| (3 / 1) + 2 |} =>* {| 1 {| [] + 2 |} {| 3 / [] |} |} .
search[1] {| 1 {| [] + 2 |} {| 3 / [] |} |} =>* {| (3 / 1) + 2 |} .
```

also succeed, showing that our second attempt to use the airlock for evaluation strategies fails, too.

One could also try the following, which should also succeed (proof is the searches above):

```
search[1] {| 3 / (1 + 2) |} =>* {| (3 / 1) + 2 |} .
search[1] {| (3 / 1) + 2 |} =>* {| 3 / (1 + 2) |} .
```

```
mod CHAM is
  sorts Molecule Solution Bag{Molecule} .
  subsort Solution < Molecule < Bag{Molecule} .
  op empty : -> Bag{Molecule} .
  op _#_ : Bag{Molecule} Bag{Molecule} -> Bag{Molecule} [assoc comm id: empty] .
  op {|_|} : Bag{Molecule} -> Solution .
  op _<|_ : Molecule Solution -> Molecule [prec 110] .
--- The airlock is unnecessary in this particular example.
--- We keep it, though, just in case will be needed as we extend the language.
--- We comment the two airlock rules below out to avoid non-termination.
--- Otherwise we would have to use slower search commands instead of rewrite.
---  var M : Molecule .  var Ms : Bag{Molecule} .
---  rl {| M # Ms |} => {| M <| {| Ms |} |} .
---  rl {| M <| {| Ms |} |} => {| M # Ms |} .
endm
```

Figure 3.50: Generic representation of the CHAM in Maude.

However, on our machine (Linux, 2.4GHz, 8GB memory) Maude 2 ran out of memory after several minutes when asked to execute any of the two search commands above.

Figures 3.50, 3.51 and 3.52 give a correct Maude representation of CHAM(IMP), based on the rewrite logic theory $\mathcal{R}_{\text{CHAM(IMP)}}$ in Figure 3.48. Three important observations were the guiding factors of our Maude semantics of CHAM(IMP):

1. While the approach to syntax in Figure 3.49 elegantly allows to include the syntax of evaluation contexts into the syntax of molecules by simply defining $\Box$ as a molecular construct, unfortunately it still requires us to redefine the entire syntax of IMP into specific constructs for molecules. This is inconvenient at best. We can do better by simply subsorting to `Molecule` all those syntactic categories that the approach in Figure 3.49 would collapse into `Molecule`. This way, any fragment of IMP code parses to a subsort of `Molecule`, so in particular to `Molecule`. Unfortunately, evaluation contexts are now ill-formed; for example, $\Box + 2$ attempts to sum a molecule with an integer, which does not parse. To fix this, we define a new sort for the $\Box$ constant, say `Hole`, and declare it as a subsort of each IMP syntactic category that is subsorted to `Molecule`. In particular, `Hole` is a subsort of `AExp`, so $\Box + 2$ parses to `AExp`.

2. As discussed above, most of CHAM's heating/cooling rules are reversible and thus, when regarded as rewrite rules, lead to non-termination. To avoid non-termination, we restrict the heating/cooling rules so that heating only applies when the heated subterm has computational contents (i.e., it is not a result) while cooling only applies when the cooled term is completely processed (i.e., it is a result). This way, the heating and the cooling rules are applied in complementary situations, in particular they are not reversible anymore, thus avoiding non-termination. To achieve this, we introduce a subsort `Result` of `Molecule`, together with subsorts of it corresponding to each syntactic category of IMP as well as with an explicit declaration of IMP's results as appropriate terms of corresponding result sort.

3. The CHAM's philosophy is to represent data, in particular program states, using its builtin support for solutions as multisets of molecules, and to use airlock operations to extract pieces of data from such solutions whenever needed. This philosophy is justified both by chemical and by mathematical intuitions, namely that one needs an additional step to observe inside a solution and, respectively, that multiset matching is a complex operation (it is actually an intractable problem) whose complexity

```
mod IMP-HEATING-COOLING-CHAM is including IMP-SYNTAX + CHAM .
  sorts Hole ResultAExp ResultBExp ResultStmt Result .
  subsorts Hole < AExp BExp Stmt < Molecule .
  subsorts ResultAExp ResultBExp ResultStmt < Result < Molecule .
  subsorts Int < ResultAExp < AExp .
  subsorts Bool < ResultBExp < BExp .
  subsorts ResultStmt < Block .
  op {} : -> ResultStmt [ditto] .

  op [] : -> Hole .
  op _~>_ : Molecule Molecule -> Molecule [gather(e E) prec 120] .

  var X : Id .  var C : [Molecule] .  var A A1 A2 : AExp .  var R R1 R2 : Result .
  var B B1 B2 : BExp .  var I I1 I2 : Int .  var S S1 S2 : Stmt .  var Ms : Bag{Molecule} .

 crl {| (A1 + A2 ~> C) # Ms |} => {| (A1 ~> [] + A2 ~> C) # Ms |} if notBool(A1 :: Result) .
  rl {| (R1 ~> [] + A2 ~> C) # Ms |} => {| (R1 + A2 ~> C) # Ms |} .

 crl {| (A1 + A2 ~> C) # Ms |} => {| (A2 ~> A1 + [] ~> C) # Ms |} if notBool(A2 :: Result) .
  rl {| (R2 ~> A1 + [] ~> C) # Ms |} => {| (A1 + R2 ~> C) # Ms |} .

 crl {| (A1 / A2 ~> C) # Ms |} => {| (A1 ~> [] / A2 ~> C) # Ms |} if notBool(A1 :: Result) .
  rl {| (R1 ~> [] / A2 ~> C) # Ms |} => {| (R1 / A2 ~> C) # Ms |} .

 crl {| (A1 / A2 ~> C) # Ms |} => {| (A2 ~> A1 / [] ~> C) # Ms |} if notBool(A2 :: Result) .
  rl {| (R2 ~> A1 / [] ~> C) # Ms |} => {| (A1 / R2 ~> C) # Ms |} .

 crl {| (A1 <= A2 ~> C) # Ms |} => {| (A1 ~> [] <= A2 ~> C) # Ms |} if notBool(A1 :: Result) .
  rl {| (R1 ~> [] <= A2 ~> C) # Ms |} => {| (R1 <= A2 ~> C) # Ms |} .

 crl {| (R1 <= A2 ~> C) # Ms |} => {| (A2 ~> R1 <= [] ~> C) # Ms |} if notBool(A2 :: Result) .
  rl {| (R2 ~> R1 <= [] ~> C) # Ms |} => {| (R1 <= R2 ~> C) # Ms |} .

 crl {| (! B ~> C) # Ms |} => {| (B ~> ! [] ~> C) # Ms |} if notBool(B :: Result) .
  rl {| (R ~> ! [] ~> C) # Ms |} => {| (! R ~> C) # Ms |} .

 crl {| (B1 && B2 ~> C) # Ms |} => {| (B1 ~> [] && B2 ~> C) # Ms |} if notBool(B1 :: Result) .
  rl {| (R1 ~> [] && B2 ~> C) # Ms |} => {| (R1 && B2 ~> C) # Ms |} .

 crl {| (X = A ; ~> C) # Ms |} => {| (A ~> X = [] ; ~> C) # Ms |} if notBool(A :: Result) .
  rl {| (R ~> X = [] ; ~> C) # Ms |} => {| (X = R ; ~> C) # Ms |} .

 crl {| (S1 S2 ~> C) # Ms |} => {| (S1 ~> [] S2 ~> C) # Ms |} if notBool(S1 :: Result) .
  rl {| (R1 ~> [] S2 ~> C) # Ms |} => {| (R1 S2 ~> C) # Ms |} .

 crl {| S # Ms |} => {| (S ~> []) # Ms |} if notBool(S :: Result) .
  rl {| (R ~> []) # Ms |} => {| R # Ms |} .

 crl {| (if (B) S1 else S2 ~> C) # Ms |} => {| (B ~> if ([]) S1 else S2 ~> C) # Ms |}
  if notBool(B :: Result) .
  rl {| (R ~> if ([]) S1 else S2 ~> C) # Ms |} => {| (if (R) S1 else S2 ~> C) # Ms |} .
endm
```

Figure 3.51: Efficient heating-cooling rules for IMP in Maude.

```
mod IMP-SEMANTICS-CHAM is including IMP-HEATING-COOLING-CHAM + STATE .
  subsort Pgm State < Molecule .
  var X : Id .  var Xl : List{Id} .  var C : Molecule .  var Ms : Bag{Molecule} .
  var Sigma : State .  var B B2 : BExp .  var I J I1 I2 : Int .  var S S1 S2 : Stmt .
  rl {| {| X ~> C |} # {| X |-> I & Sigma |} # Ms |}
  => {| {| I ~> C |} # {| X |-> I & Sigma |} # Ms |} .
  rl {| (I1 + I2 ~> C) # Ms |} => {| (I1 +Int I2 ~> C) # Ms |} .
 crl {| (I1 / I2 ~> C) # Ms |} => {| (I1 /Int I2 ~> C) # Ms |} if I2 =/=Bool 0 .
  rl {| (I1 <= I2 ~> C) # Ms |} => {| (I1 <=Int I2 ~> C) # Ms |} .
  rl {| (! true ~> C) # Ms |} => {| (false ~> C) # Ms |} .
  rl {| (! false ~> C) # Ms |} => {| (true ~> C) # Ms |} .
  rl {| (true && B2 ~> C) # Ms |} => {| (B2 ~> C) # Ms |} .
  rl {| (false && B2 ~> C) # Ms |} => {| (false ~> C) # Ms |} .
  rl {| ({S} ~> C) # Ms |} => {| (S ~> C) # Ms |} .
  rl {| {| X = I ; ~> C |} # {| X |-> J & Sigma |} # Ms |}
  => {| {| {} ~> C |} # {| X |-> I & Sigma |} # Ms |} .
  rl {| ({} S2 ~> C) # Ms |} => {| (S2 ~> C) # Ms |} .
  rl {| (if (true) S1 else S2 ~> C) # Ms |} => {| (S1 ~> C) # Ms |} .
  rl {| (if (false) S1 else S2 ~> C) # Ms |} => {| (S2 ~> C) # Ms |} .
  rl {| (while (B) S ~> C) # Ms |} => {| (if (B) {S while (B) S} else {} ~> C) # Ms |} .
  rl {| (int Xl ; S) # Ms |} => {| {| S |} # {| Xl |-> 0 |} # Ms |} .
endm
```

Figure 3.52: The CHAM of IMP in Maude.

cannot be simply "swept under the carpet". While we agree with these justifications for the airlock operator, one should also note that impressive progress has been made in the last two decades, after the proposal of the chemical abstract machine, in terms of multiset matching. For example, languages like Maude build upon very well-engineered multiset matching techniques. We believe that these recent developments justify us, at least in practical language definitions, to replace the expensive airlock operation of the CHAM with the more available and efficient multiset matching of Maude.

Figure 3.50 shows the generic CHAM syntax that we extend in order to define CHAM(IMP). Since we use Maude's multiset matching instead of state airlock and since we cannot use the airlock for evaluation strategies either, there is effectively no need for the airlock in our Maude definition of CHAM(IMP). Moreover, since the airlock rules are reversible, their introduction would yield non-termination. Consequently, we have plenty of reasons to eliminate them, which is reflected in our `CHAM` module in Figure 3.50. The Maude module in Figure 3.51 defines the evaluation strategies of IMP's constructs and should be now clear: in addition to the syntactic details discussed above, it simply gives the Maude representation of the heating/cooling rules in Figure 3.48. Finally, the Maude module in Figure 3.52 implements the semantic rules and the state initialization heating rule in Figure 3.48, replacing the state airlock operation by multiset matching.

To test the heating/cooling rules, one can write Maude commands such as the two search commands below, asking Maude to search for all heatings of a given syntactic molecule (the sort of *X*, *Id*, is already declared in the last module):

```
search {| X = 3 / (X + 2) ; |} =>! Sol:Solution .
search {| X = (Y:Id + Z:Id) / (X + 2) ; |} =>! Sol:Solution .
```

The former gives only one solution, because heating can only take place on non-result subexpressions

```
Solution 1 (state 4)
states: 5  rewrites: 22 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| X ~> [] + 2 ~> 3 / [] ~> X = [] ; ~> [] |}
```

but the latter gives three solutions:

```
Solution 1 (state 5)
states: 8  rewrites: 31 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| Y:Id ~> [] + Z:Id ~> [] / (X + 2) ~> X = [] ; ~> [] |}

Solution 2 (state 6)
states: 8  rewrites: 31 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| Z:Id ~> Y:Id + [] ~> [] / (X + 2) ~> X = [] ; ~> [] |}

Solution 3 (state 7)
states: 8  rewrites: 31 in ... cpu (... real) (~ rewrites/second)
Sol:Solution --> {| X ~> [] + 2 ~> (Y:Id + Z:Id) / [] ~> X = [] ; ~> [] |}
```

Each of the solutions represents a completely heated term (we used `=>!` in the search commands) and corresponds to a particular order of evaluation of the subexpression in question. The three solutions of the second search command above reflect all possible orders of evaluation allowed by our Maude semantics of CHAM(IMP). Interestingly, there is no order in which X is looked up in between Y and Z. This is not a problem for IMP, but it may result in loss of behaviors for IMP++ programs. Indeed, it may be that other threads modify the values of X, Y, and Z while the expression above is evaluated by another thread in such a way that behaviors are lost if X is not allowed to be looked up between Y and Z. Consequently, our orientation of the heating/cooling rules came at a price: we lost the fully non-deterministic order of evaluation of the arguments of strict operators; what we obtained is a *non-deterministic choice* evaluation strategy (an order of evaluation is non-deterministically chosen and cannot be changed during the evaluation of the expression—this is discussed in more depth in Section 3.5).

Maude can now act as an execution engine for CHAM(IMP). For example, the Maude command

```
rewrite {| sumPgm |} .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form:

```
rewrite in TEST : {| sumPgm |} .
rewrites: 7543 in ... cpu (... real) (... rewrites/second)
result Solution: {| {| {} |} # {| n |-> 0 & s |-> 5050 |} |}
```

Like in the previous Maude semantics, one can also search for all possible behaviors of a program using `search` commands such as

```
search {| sumPgm |} =>! Sol:Solution .
```

Like before, only one behavior will be discovered (IMP is deterministic so far). However, an unexpectedly large number of states is generated, 4119 versus the 1709 states generated by the previous small-step semantics), mainly due to the multiple ways to apply the heating/cooling rules:

```
Solution 1 (state 4118)
states: 4119  rewrites: 12658 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| {| {} |} # {| n |-> 0 & s |-> 5050 |} |}
```

3.8.4 Notes

The chemical abstract machine, abbreviated CHAM, was introduced by Berry and Boudol in 1990 [8, 9]. In spite of its operational feel, the CHAM should not be mistakenly taken for a variant of (small-step) SOS. In

fact, Berry and Boudol presented the CHAM as an *alternative* to SOS, to address a number of limitations inherent to SOS, particularly its lack of true concurrency and what they called SOS' "rigidity to syntax". The basic metaphor giving its name to the CHAM was inspired by Banâtre and Le Mètayer's GAMMA language [4, 5, 6], which was the first to view a distributed state as a solution in which many molecules float, and the first to understand concurrent transitions as reactions that can occur simultaneously in many points of the solution. GAMMA was proposed as a highly-parallel programming language, together with a stepwise program derivation approach that allows to develop provably correct GAMMA programs. However, following the stepwise derivation approach to writing GAMMA programs is not as straightforward as writing CHAM rules. Moreover, CHAM's nesting of solutions allows for structurally more elaborate encodings of data and in particular for more computational locality than GAMMA. Also, the CHAM appears to be more suitable as a framework for defining semantics of programming languages than the GAMMA language; the latter was mainly conceived as a programming language itself, suitable for executing parallel programs on parallel machines [3], rather than as a semantic framework.

The distinction between heating, cooling and reaction rules in CHAM is in general left to the user. There are no well-accepted criteria and/or principles stating when a rule should be in one category or another. For example, should a rule that cleans up a solution by removing residue molecules be a heating or a colling rule? We prefer to think of it as a cooling rule, because it falls under the broad category of rules which "structurally rearrange the solution after reactions take place" which we methodologically decided to call cooling rules. However, the authors of the CHAM prefer to consider such rules to be heating rules, with the intuition that "the residue molecules evaporate when heated" [9]. While the distinction between heating and cooling rules may have a flavor of subjectivity, the distinction between actual reaction rules and heating/cooling rules is more important because it gives the computational granularity of one's CHAM. Indeed, it is common to abstract away the heating/cooling steps in a CHAM rewriting sequence as internal steps and then define various relations of interest on the remaining reaction steps possibly relating different CHAMS, such as behavioral equivalence, simulation and/or bisimulation relations [8, 9].

The technique we used in this section to reversibly and possibly non-deterministically sequentialize the program syntax by heating/cooling it into a list of computational tasks was borrowed from the K framework [68, 66, 45, 64] (also Section 3.12). This mechanism is also reminiscent of Danvy and Nielsen's refocusing technique [19, 20], used to execute reduction semantics with evaluation contexts by decomposing evaluation contexts into stacks and then only incrementally modifying these stacks during the reduction process. Our representation of the CHAM into rewrite logic was inspired from related representations by Șerbănuță *et al.* [74] and by Meseguer[12] [42]. However, our representation differs in that it enforces the CHAM rewriting to only take place in solutions, this way being completely faithful to the intended meaning of the CHAM reactions, while the representations in [74, 42] are slightly more permissive, allowing rewrites to take place everywhere rules match; as explained, this is relevant only when the left-hand-side of the rule contains precisely one molecule.

We conclude this section with a note on the concurrency of CHAM rewriting. As already explained, we are not aware of any formal definition of a CHAM rewriting relation that captures the truly concurrent computation advocated by the CHAM. This is unfortunate, because its concurrency potential is one of the most appealing aspects of the CHAM. Moreover, as already discussed (after Theorem 21), our rewrite logic representation of the CHAM is faithful only with respect to one non-concurrent step and interleaves steps taking place within the same solutions no matter whether they involve common molecules or not, so we cannot borrow rewrite logic's concurrency to obtain a concurrent semantics for the CHAM. What can be done,

[12]Meseguer [42] was published in the same volume of the Journal of Theoretical Computer Science as the CHAM extended paper [9] (the first CHAM paper [8] was published two years before, in 1990, same as the first papers on rewrite logic [40, 39, 41]).

however, is to attempt a different representation of the CHAM into rewrite logic based on ideas proposed by Meseguer in [43] to capture (a limited form of) graph rewriting by means of equational encodings. The encoding in [43] is theoretically important, but, unfortunately, yields rewrite logic theories which are not feasible in practice using the current implementation of Maude.

3.8.5 Exercises

Exercise 168. *For any CHAM, any molecules $mol_1, \ldots, mol_k$, and any $1 \leq i \leq k$, the sequents*

$$\text{CHAM} \vdash \{\!|mol_1 \ \ldots \ mol_k|\!\} \leftrightarrow \{\!|mol_1 \rhd \{\!|mol_2 \ \ldots mol_i|\!\} \ mol_{i+1} \ldots mol_k|\!\}$$

are derivable, where if $i = 1$ then the bag $mol_2 \ \ldots mol_i$ is the empty bag.

Exercise 169. *Modify the CHAM semantics of* IMP *in Figure 3.45 to use a state data-structure as we did in the previous semantics instead of representing the state as a solution of binding molecules.*

Exercise 170. *Add a cleanup (cooling) rule to the CHAM semantics of* IMP *in Figure 3.45 to remove the useless syntactic subsolution when the computation is terminated. The resulting solution should only contain a state (i.e., it should have the form $\{\!|\sigma|\!\}$, and not $\{\!|\{\!|\sigma|\!\}|\!\}$ or $\{\!|\cdot \rhd \{\!|\sigma|\!\}|\!\}$).*

Exercise 171. *Modify the CHAM semantics of* IMP *in Figures 3.44 and 3.45 so that / short-circuits when the numerator evaluates to* 0.
Hint: One may need to make one of the heating/cooling rules for / conditional.

Exercise 172. *Modify the CHAM semantics of* IMP *in Figures 3.44 and 3.45 so that conjunction is non-deterministically strict in both its arguments.*

Exercise 173. *Same as Exercise 84, but for CHAM instead of big-step SOS: add variable increment to* IMP*, like in Section 3.8.2*
★ *Like for the Maude definition of big-step SOS and unlike for the Maude definitions of small-step SOS, MSOS and reduction semantics with evaluation contexts above, the resulting Maude definition can only exhibit three behaviors (instead of five) of the program* `nondet++Pgm` *in Exercise 84. This limitation is due to our decision (in Section 3.8) to only heat on non-results and cool on results when implementing CHAMs into Maude. This way, the resulting Maude specifications are executable at the expense at losing some of the behaviors due to non-deterministic evaluation strategies.*

Exercise 174. *Same as Exercise 88, but for the CHAM instead of big-step SOS: add input/output to* IMP*, like in Section 3.8.2.*

Exercise 175. *Same as Exercise 92, but for the CHAM instead of big-step SOS: add abrupt termination to* IMP*, like in Section 3.8.2.*

Exercise 176. *Same as Exercise 104, but for the CHAM instead of small-step SOS: add dynamic threads to* IMP*, like in Section 3.8.2.*

Exercise* 177. *This exercise asks to define* IMP++ *in CHAM, in various ways. Specifically, redo Exercises 114, 115, 116, 117, and 118, but for the CHAM of* IMP++ *discussed in Section 3.8.2 instead of its small-step SOS in Section 3.5.6.*